

REMARKS

The application has been reviewed in light of the Office Action mailed April 19, 2004. At the time of the Office Action, Claims 1-4 were pending in this application. Claims 1-4 were rejected in the Office Action. Claims 1-4 have been canceled and claims 5-20 have been added.

Declaration Defective

The Examiner indicated that the oath/declaration was defective because the title on the declaration did not match that of the application. To remedy the problem, the title of the present application has been amended to conform to the title of the invention in the declaration.

Objections to the Specification

The Examiner objected to the disclosure because of a variety of informalities. Specifically, the appendixes as submitted originally were improper. As helpfully suggested by the Examiner, the contents of the appendix have been moved into the specification. As the appendix was quite lengthy, applicant herewith submits a substitute specification under 37 C.F.R. 1.125(c). A marked up version of the specification is included with this response.

Rejections to the Claims

The Examiner rejected claim 4 under 35 U.S.C. 112, second paragraph. In response to the rejection, Applicants have canceled claim 4. Withdrawal of the rejection is respectfully requested.

Rejections to the Claims under 35 U.S.C. 102(b)

The Examiner rejected claims 1-4 under 35 U.S.C. 102(b) as being anticipate by Carlson, et al., U.S. Patent No. 4,090,250 ("Carlson"). In view of the rejection under Section

112, the Applicants have canceled claims 1-4 and have added new claims 5-20 that more particularly point out and distinctly claim the subject matter that the Applicants regard as their invention. Reconsideration and withdrawal of the rejection are respectfully requested.

Applicants respectfully submit that no amendments have been made to the pending claims for the purpose of overcoming any prior art rejections that would restrict the literal scope of the claims or equivalents thereof.

Applicants respectfully request that the amendments submitted herein be entered, and further request reconsideration in light of the amendments and remarks contained herein.

Applicants respectfully request withdrawal of all objections and rejections, and that there be an early notice of allowance.

SUMMARY

In light of the above amendments and remarks Applicants respectfully submit that the application is now in condition for allowance and early notice of the same is earnestly solicited. Should the Examiner have any questions, comments or suggestions in furtherance of the prosecution of this application, the Examiner is invited to contact the attorney of record by telephone or facsimile.

Applicants believe that there are no fees due in association with the filing of this Response, other than the three-month extension of time. However, should the Commissioner deem that any additional fees are due, including any fees for extensions of time, Applicants respectfully request that the Commissioner accept this as a Petition Therefor, and direct that any and all fees due are charged to Baker Botts L.L.P. **Deposit Account No. 02-0383**, (*formerly Baker & Botts, L.L.P.*) **Order Number 068354.1444**.

Respectfully submitted,

BAKER BOTTS L.L.P. (023640)

By: 

Ronald L. Chichester

Reg. No. 36,765

One Shell Plaza

910 Louisiana Street

Houston, Texas 77002-4995

Telephone: 713.229.1341

Facsimile: 713.229.7741

E-Mail: Ronald.Chichester@bakerbotts.com

ATTORNEY FOR APPLICANTS

October 19, 2004

DIGITAL SIGNAL CONTROLLER INSTRUCTION SET AND
ARCHITECTURE UNITED STATES PATENT APPLICATION

CERTIFICATE OF MAILING VIA EXPRESS MAIL

I HEREBY CERTIFY THAT THIS PAPER OR FEE IS BEING
DEPOSITED WITH THE UNITED STATES POSTAL SERVICE
"EXPRESS MAIL POST OFFICE TO ADDRESSEE"
SERVICE UNDER 37 C.F.R. 1.10 ON THE DATE INDICATED
BELOW, ADDRESSED TO:

MAIL STOP PROVISIONAL PATENT
APPLICATION

HONORABLE COMMISSIONER OF PATENTS
P.O. BOX 1450
ALEXANDRIA, VA 22313-1450

PAUL N. KATZ REG. No. 35,917

MAILING DATE: OCTOBER 19, 2004

EXPRESS MAIL LABEL: EV448726915 US

APPLICATION FOR LETTERS PATENT

FOR

MICROCONTROLLER INSTRUCTION SET

Inventor(s): Catherwood, Michael L.; Boles, Brian; Bowling, Stephen A.; Conner, Joshua
M.; Drake, Rodney; Elliot, John; Fall, Brian Neil; Grosbach, James H.;
Kuhrt, Tracy Ann; McCarthy, Guy; Muro, Manuel JR.; Pyska, Michael;
Triece, Joseph W.

Assignee: Microchip Technology Inc.

Attorney: Ronald L. Chichester of Baker Botts L.L.P.

Docket No.: 068354.1530

MICROCONTROLLER INSTRUCTION SET

CROSS REFERENCE TO RELATED APPLICATIONS

[0001] ~~[0001]~~ This application is a continuation-in-part of, and claims priority to, U.S. Patent Application Serial Number 09/870,457 which was filed on June 1, 2001 by the same inventors and assigned to the same entity, and is herein incorporated by reference for all purposes. This application is related to the following applications: U.S. application for "Repeat Instruction with Interrupt" on ~~Jan.~~ June 1, 2001 by M. Catherwood, et al. (MTI-1665); U.S. application for "Low Overhead Interrupt" on ~~Jan.~~ June 1, 2001 by M. Catherwood, et al. (MTI-1666); U.S. application for "Find First Bit Value Instructions" on ~~Jan.~~ June 1, 2001 by M. Catherwood (MTI-1667); U.S. application for "Bit Replacement and Extraction Instructions" on ~~Jan.~~ June 1, 2001 by B. Boles, et al. (MTI-1668); U.S. application for "Shadow Register Array Control Instructions" on ~~Jan.~~ June 1, 2001 by M. Catherwood, et al. (MTI-1669); U.S. application for "Multi-Precision Barrel Shifting" on ~~Jan.~~ June 1, 2001 by J. Conner, et al. (MTI-1670); U.S. application for "Dynamically Reconfigurable Data Space" on ~~Jan.~~ June 1, 2001 by M. Catherwood, et al. (MTI-1735); U.S. application for "Modified Harvard Architecture Processor Having Data Memory Space Mapped to Program Memory Space" on ~~Jan.~~ June 1, 2001 by J. Grosbach, et al. (MTI-1736); U.S. application for "Modified Harvard Architecture Processor Having Data Memory Space Mapped to Program Memory Space with Erroneous Execution Protection" on ~~Jan.~~ June 1, 2001 by M. Catherwood (MTI-1737); U.S. application for "Dual Mode Arithmetic Saturation Processing" on ~~Jan.~~ June 1, 2001 by M. Catherwood (MTI-1738); U.S. application for "Compatible Effective Addressing With a Dynamically Reconfigurable Data Space Word Width" on ~~Jan.~~ June 1, 2001 by M.

Catherwood, et al. (MTI-1739); U.S. application for "Maximally Negative Signed Fractional Number Multiplication" on ~~Jun.~~**June** 1, 2001 by M. Catherwood (MTI-1754); U.S. application for "Euclidean Distance Instructions" on ~~Jun.~~**June** 1, 2001 by M. Catherwood (MTI-1755); U.S. application for "Sticky Z Bit" on ~~Jun.~~**June** 1, 2001 by J. Elliot (MTI-1756); U.S. application for "Variable Cycle Interrupt Disabling" on ~~Jun.~~**June** 1, 2001 by B. Boles, et al. (MTI-1757); U.S. application for "Register Pointer Trap" on ~~Jun.~~**June** 1, 2001 by M. Catherwood (MTI-1758); U.S. application for "Modulo Addressing Based on Absolute Offset" on ~~Jun.~~**June** 1, 2001 by M. Catherwood (MTI-1759); U.S. application for "Dual Dead Time Unit for PWM Module" on ~~Jun.~~**June** 1, 2001 by S. Bowling (MTI-1789); U.S. application for "Fault Pin Priority" on ~~Jun.~~**June** 1, 2001 by S. Bowling (MTI-1790); U.S. application for "Extended Resolution Mode for PWM Module" on ~~Jun.~~**June** 1, 2001 by S. Bowling (MTI-1791); U.S. application for "Configuration Fuses for Setting PWM Options" on ~~Jun.~~**June** 1, 2001 by S. Bowling (MTI-1792); U.S. application for "Automatic A/D Sample Triggering" on ~~Jun.~~**June** 1, 2001 by B. Boles (MTI-1794); U.S. application for "Reduced Power Option" on ~~Jun.~~**June** 1, 2001 by M. Catherwood (MTI-1796) which are all hereby incorporated herein by reference for all purposes.

FIELD OF THE INVENTION

[0002] ~~[0002]~~ The present invention relates generally to processor instruction sets and, more particularly, to an instruction set for processing micro-controller type instructions and digital signal processor instructions from a single instruction stream.

BACKGROUND OF THE INVENTION

[0003] ~~[0003]~~ Processors, including microprocessors, digital signal processors and microcontrollers, operate by running software programs that are embodied in one or more series

of instructions stored in a memory. The processors run the software by fetching the instructions from the series of instructions, decoding the instructions and executing them.

[0004] ~~[0004]~~ In addition to program instructions, data is also stored in memory that is accessible by the processor. Generally, the program instructions process data by accessing data in memory, modifying the data and storing the modified data into memory.

[0005] ~~[0005]~~ The instructions themselves also control the sequence of functions that the processor performs and the order in which the processor fetches and executes the instructions. For example, the order for fetching and executing each instruction may be inherent in the order of the instructions within the series. Alternatively, instructions such as branch instructions, conditional branch instructions, subroutine calls and other flow control instructions may cause instructions to be fetched and executed out of the inherent order of the instruction series.

[0006] ~~[0006]~~ The program instructions that comprise a software program are taken from an instruction set that is designed for each processor. The instruction set includes a plurality of instructions, each of which specifies operations of one or more functional components of the processor. The instructions are decoded in an instruction decoder which generates control signals distributed to the functional components of the processor to perform the operation(s) specified in the instruction.

[0007] ~~[0007]~~ The instruction set itself, in terms of breadth, flexibility and simplicity dictates the ease with which programmers may generate programs. The instruction set also reflects the processor architecture and accordingly the functional and performance capability of the processor.

[0008] ~~[0008]~~ There is a need for a processor and an instruction set that includes a robust and an efficient set of instructions for a wide variety of applications. Given the rapid growth of

digital signal processing (DSP) applications, there is a further need for an instruction set that incorporates DSP type instructions and micro-controller type instructions. There is a further need to provide processor having a tightly coupled DSP engine and a microcontroller arithmetic logic unit (ALU) for many types of applications conventionally handled separately by either a microcontroller or a digital signal processor, including motor control, soft modems, automotive body computers, speech recognition, echo cancellation and fingerprint recognition.

SUMMARY OF THE INVENTION

[0009] ~~[0009]~~ According to embodiments of the present invention, an instruction set is provided that features ninety four instructions and eleven address modes to deliver a mixture of flexible micro-controller like instructions and specialized digital signal processor (DSP) instructions that execute from a single instruction stream.

[0010] ~~[0010]~~ According to an embodiment of the present invention, a processor executes instructions within the designated instruction set. The processor includes a program memory, a program counter, registers and at least one execution unit. The program memory stores program instructions, including instructions from the designated instruction set. The program counter determines the current instruction for processing. The registers store operand data specified by the program instructions and the execution unit(s) execute the current instruction. The execution unit may include a DSP engine and arithmetic logic unit. Each designated instruction is identified to the processor by designated encoding and to programmers by a designated mnemonic.

BRIEF DESCRIPTION OF THE FIGURES DRAWINGS

[0011] ~~[0011]~~ The above described features A more complete understanding of the present disclosure and advantages of the present invention will be more fully appreciated with

reference to the detailed thereof may be acquired by referring to the following description and appended figures in which taken in conjunction with the accompanying drawings, wherein:

[0012] ~~[0012]~~ FIG. Figure 1 depicts a functional block diagram of an embodiment of a processor chip within which embodiments of the present invention may find application.

[0013] ~~[0013]~~ FIG. Figure 2 depicts a functional block diagram of a data busing scheme for use in a processor, which has a microcontroller and a digital signal processing engine, within which embodiments of the present invention may find application.

[0014] ~~[0014]~~ FIG. Figure 3 depicts a functional block diagram of a digital signal processor ("DSP") engine according to an embodiment of the present invention. disclosure.

[0015] ~~[0015]~~ FIGS. Figures 4A-4E depict five different instruction flow types according to embodiments of the present invention. disclosure.

[0016] ~~[0016]~~ FIG. Figure 5 depicts a programmer's model of the processor according to an embodiment of the present invention. disclosure.

[0017] Figure 4B is a block diagram illustrating an Instruction Pipeline Flow - 1
Word 2 Cycle according to an embodiment of the present disclosure.

[0018] Figure 4C is a block diagram illustrating an Instruction Pipeline Flow - 1
Word 2 Cycle Table Operations according to an embodiment of the present disclosure.

[0019] Figure 4D is a block diagram illustrating an Instruction Pipeline Flow - 2
Word 2 Cycle GOTO, CALL according to an embodiment of the present disclosure.

[0020] Figure 4E is a block diagram illustrating an Instruction Pipeline Flow - 2
word 2 cycle DO, DOW according to an embodiment of the present disclosure.

[0021] Figure 5 is a block diagram illustrating a Programmers model according to
an embodiment of the present disclosure.

[0022] Figure 6 is a block diagram illustrating a Program Memory Addressing Scheme according to an embodiment of the present disclosure.

[0023] Figure 7 is a block diagram illustrating a "CALL lit23" Map to the Program Counter according to an embodiment of the present disclosure.

[0024] Figure 8 is a block diagram illustrating a "BRA SLIT16" Map to the Program Counter according to an embodiment of the present disclosure.

[0025] Figure 9 is a block diagram illustrating a "GOTO Wn" Map to the Program Counter according to an embodiment of the present disclosure.

[0026] Figure 10 is a block diagram illustrating a "BRA Wn" Map to the Program Counter according to an embodiment of the present disclosure.

[0027] Figure 11 is a block diagram illustrating a Data Alignment in Memory according to an embodiment of the present disclosure.

[0028] Figure 12 is a block diagram illustrating a MOV.D operation according to an embodiment of the present disclosure.

[0029] Figure 13 is a block diagram illustrating a MOV.Q operation according to an embodiment of the present disclosure.

[0030] Figure 14 is a block diagram illustrating a stack at the beginning of a calling sequence according to an embodiment of the present disclosure.

[0031] Figure 15 is a block diagram illustrating a stack at the entry to a routine according to an embodiment of the present disclosure.

[0032] Figure 16 is a block diagram illustrating a stack after a LNK instruction according to an embodiment of the present disclosure.

[0033] Figure 17 is a block diagram illustrating a Multi-Word Left Shift by 4 Instruction Execution according to an embodiment of the present disclosure.

[0034] Figure 18 is a block diagram illustrating a Multi-Word Left Shift by 20 Instruction Execution according to an embodiment of the present disclosure.

[0035] Figure 19 is a block diagram illustrating a Multi-Word Right Shift by 4 Instruction Execution according to an embodiment of the present disclosure.

[0036] Figure 20 is a block diagram illustrating a Multi-Word Right Shift by 20 Instruction Execution according to an embodiment of the present disclosure.

[0037] Figure 21 is a block diagram illustrating a 16-Bit integer and fractional modes according to an embodiment of the present disclosure.

[0038] Figure 22 is a block diagram illustrating a DO operation according to an embodiment of the present disclosure.

[0039] Figure 23 is a block diagram illustrating an alternate embodiment of the DO operation according to an embodiment of the present disclosure.

[0040] Figure 24 is a block diagram illustrating a Register Direct addressing mode according to an embodiment of the present disclosure.

[0041] Figure 25 is a block diagram illustrating an alternate Register Indirect addressing mode according to an embodiment of the present disclosure.

[0042] Figure 26 is a block diagram illustrating a Register Indirect with Post-Decrement addressing mode according to an embodiment of the present disclosure.

[0043] Figure 27 is a block diagram illustrating a Register Indirect with Post-Increment addressing mode according to an embodiment of the present disclosure.

[0044] Figure 28 is a block diagram illustrating a Register Indirect with Pre-Decrement addressing mode according to an embodiment of the present disclosure.

[0045] Figure 29 Register Indirect with Pre-Increment Addressing mode according to an embodiment of the present disclosure.

[0046] Figure 30 is a block diagram illustrating a Register Direct with 5-bit signed Literal Operation mode according to an embodiment of the present disclosure.

[0047] Figure 31 is a block diagram illustrating a Register Direct, Operand Source mode according to an embodiment of the present disclosure.

[0048] Figure 32 is a block diagram illustrating a Register Indirect, Result Destination mode according to an embodiment of the present disclosure.

[0049] Figure 33 is a block diagram illustrating a Register Indirect, Operand Source mode according to an embodiment of the present disclosure.

[0050] Figure 34 is a block diagram illustrating a Register Indirect, Result Destination mode according to an embodiment of the present disclosure.

[0051] Figure 35 is a block diagram illustrating a Register Indirect with Post Decrement, Source Operand mode according to an embodiment of the present disclosure.

[0052] Figure 36 is a block diagram illustrating a Register Indirect with Post Decrement, Result Destination mode according to an embodiment of the present disclosure.

[0053] Figure 37 is a block diagram illustrating a Register Indirect with Post Increment, Operand Source mode according to an embodiment of the present disclosure.

[0054] Figure 38 is a block diagram illustrating a Register Indirect with Post Increment, Result Destination mode according to an embodiment of the present disclosure.

[0055] Figure 39 is a block diagram illustrating a Register Indirect with Pre-Decrement, Source Operand mode according to an embodiment of the present disclosure.

[0056] Figure 40 is a block diagram illustrating a Register Indirect with Pre-Decrement, Result Destination mode according to an embodiment of the present disclosure.

[0057] Figure 41 is a block diagram illustrating a Register Indirect with Pre-Increment, Source Operand mode according to an embodiment of the present disclosure.

[0058] Figure 42 is a block diagram illustrating a Register Indirect with Pre-Increment, Result Destination mode according to an embodiment of the present disclosure.

[0059] Figure 43 is a block diagram illustrating a Register Direct, Operand Source mode according to an embodiment of the present disclosure.

[0060] Figure 44 is a block diagram illustrating a Register Direct, Operand Source mode according to an embodiment of the present disclosure.

[0061] Figure 45 is a block diagram illustrating a Register Indirect, Source Operand mode according to an embodiment of the present disclosure.

[0062] Figure 46 is a block diagram illustrating a Register Indirect, Result Destination mode according to an embodiment of the present disclosure.

[0063] Figure 47 is a block diagram illustrating a Register Indirect with Post-Decrement, Source Operand mode according to an embodiment of the present disclosure.

[0064] Figure 48 is a block diagram illustrating a Register Indirect with Post-Decrement, Result Destination mode according to an embodiment of the present disclosure.

[0065] Figure 49 is a block diagram illustrating a Register Indirect with Post-Increment, Source Operand mode according to an embodiment of the present disclosure.

[0066] Figure 50 is a block diagram illustrating a Register Indirect with Post Increment, Result Destination mode according to an embodiment of the present disclosure.

[0067] Figure 51 is a block diagram illustrating a Register Indirect with Pre-Decrement, Source Operand mode according to an embodiment of the present disclosure.

[0068] Figure 52 is a block diagram illustrating a Register Indirect with Pre-Decrement, Result Destination mode according to an embodiment of the present disclosure.

[0069] Figure 53 is a block diagram illustrating a Register Indirect with Register Offset, Operand Source mode according to an embodiment of the present disclosure.

[0070] Figure 54 is a block diagram illustrating a Register Indirect with Register Offset, Result Destination mode according to an embodiment of the present disclosure.

[0071] Figure 55 is a block diagram illustrating a Register Indirect with Constant Offset, Source Operand mode according to an embodiment of the present disclosure.

[0072] Figure 56 is a block diagram illustrating a Register Indirect with Constant Offset, Result Destination mode according to an embodiment of the present disclosure.

[0073] Figure 57 is a block diagram illustrating a Register Indirect with Pre-Decrement, Source Operand mode according to an embodiment of the present disclosure.

[0074] Figure 58 is a block diagram illustrating a Register Indirect with Pre-Decrement, Result Destination mode according to an embodiment of the present disclosure.

[0075] Figure 59 is a block diagram illustrating a Register Indirect mode according to an embodiment of the present disclosure.

[0076] Figure 60 is a block diagram illustrating a Register Indirect with Post Increment mode according to an embodiment of the present disclosure.

[0077] Figure 61 is a block diagram illustrating a Register Indirect with Register Offset Operand Source mode according to an embodiment of the present disclosure.

[0078] Figure 62 is a block diagram illustrating a Register Indirect with Post Decrement mode according to an embodiment of the present disclosure.

[0079] Figure 63 is a block diagram illustrating an X AGU according to an embodiment of the present disclosure.

[0080] Figure 64 is a block diagram illustrating a Y AGU according to an embodiment of the present disclosure.

[0081] Figure 65 is a block diagram illustrating an Incrementing Buffer Modulo addressing operation according to an embodiment of the present disclosure.

[0082] Figure 66 is a block diagram illustrating a Decrementing Buffer Modulo addressing operation according to an embodiment of the present disclosure.

[0083] Figure 67 is a block diagram illustrating a Bit Reversed EA calculation according to an embodiment of the present disclosure.

[0084] Figure 68 is a block diagram illustrating a Alternative Bit Reversed EA calculation method according to an embodiment of the present disclosure.

[0085] Figure 69 is a block diagram illustrating a Bit Reversed Addressing, Source Operand mode according to an embodiment of the present disclosure.

[0086] Figure 70 is a block diagram illustrating a Bit Reversed Addressing, Destination Operand mode according to an embodiment of the present disclosure.

[0087] Figure 71 is a block diagram illustrating a Register Indirect, Table Read Operand Destination mode according to an embodiment of the present disclosure.

[0088] Figure 72 is a block diagram illustrating a Register Indirect, Table Read Operand Source mode according to an embodiment of the present disclosure.

[0089] Figure 73 is a block diagram illustrating a Register Indirect, Table Read Result Destination mode according to an embodiment of the present disclosure.

[0090] Figure 74 is a block diagram illustrating a Register Indirect with Post Decrement, Table Read Source Operand mode according to an embodiment of the present disclosure.

[0091] Figure 75 is a block diagram illustrating a Register Indirect with Post Decrement, Table Read Result Destination mode according to an embodiment of the present disclosure.

[0092] Figure 76 is a block diagram illustrating a Register Indirect with Post Increment, Table Read Operand Source mode according to an embodiment of the present disclosure.

[0093] Figure 77 is a block diagram illustrating a Register Indirect with Post Increment, Table Read Result Destination mode according to an embodiment of the present disclosure.

[0094] Figure 78 is a block diagram illustrating a Register Indirect with Pre-Decrement, Table Read Source Operand mode according to an embodiment of the present disclosure.

[0095] Figure 79 is a block diagram illustrating a Register Indirect with Pre-Decrement, Table Read Result Destination mode according to an embodiment of the present disclosure.

[0096] Figure 80 is a block diagram illustrating a Register Indirect with Pre-Increment, Table Read Source Operand mode according to an embodiment of the present disclosure.

[0097] Figure 81 is a block diagram illustrating a Register Indirect with Pre-Increment, Table Read Result Destination according to an embodiment of the present disclosure.

[0098] Figure 82 is a timing diagram illustrating a XOR, the SUBR, the SUBR, the SUB B, the SUB, the MOVE, the IOR, the AND, the ADDC and the ADD operations according to an embodiment of the present disclosure.

[0099] Figure 83 is a timing diagram illustrating a XORLS, the SUBRLS, the SUBLS, the SUBBRLS, the SUBLS, the IORLS, the ANDLS, the ADCCLS and the ADDCLS operations according to an embodiment of the present disclosure.

[0100] Figure 84 is a timing diagram illustrating a COR, the INC2, the DEC2, the DEC COM, the NEG and the NCTM operations according to an embodiment of the present disclosure.

[0101] Figure 85 is a timing diagram illustrating a ASR, the LSR, the ZE, the SE, the SL, the RLC, the RLNC, the RRC and the RRNC operation according to an embodiment of the present disclosure.

[0102] Figure 86 is a timing diagram illustrating a CPB and the CP operations according to an embodiment of the present disclosure.

[0103] Figure 87 is a timing diagram illustrating a CP1 and the CP0 operations according to an embodiment of the present disclosure.

[0104] Figure 88 is a timing diagram illustrating a CPBLS and the CPBLS operations according to an embodiment of the present disclosure.

[0105] Figure 89 is a timing diagram illustrating a XORLW, the SUBLW, the SUBBBLW, the MOVLW, the MOVL, the IORLW, the ANDLW, the ADDLW and the ADDCLW operations according to an embodiment of the present disclosure.

[0106] Figure 90 is a timing diagram illustrating a ASRF, the SLF, the LSRF, the RRNCF, the RRCE, the RLNCF, the RLCE, the XORWF, the SUBWS, the SUBBWS, the SUBFW, the SUBDFW, the MOVFW, the MOV, the IORWV, the ANDWF, the ADDWFC and the ADDWF operations according to an embodiment of the present disclosure.

[0107] Figure 91 is a timing diagram illustrating a CPFB, the CPF1, the CPF0 and the CPF operations according to an embodiment of the present disclosure.

[0108] Figure 92 is a timing diagram illustrating a INCF, the DECF, the NEGF, the SETF, the COMF and the CLRf operations according to an embodiment of the present disclosure.

[0109] Figure 93 is a timing diagram illustrating a CPFSEQ, the FPFSGT, the CPFSLT and the CPFSENE operations according to an embodiment of the present disclosure.

[0110] Figure 94 is a timing diagram illustrating an INCFSNZ, the INCFSA, the DECFSNZ, and the DECFSZ operations according to an embodiment of the present disclosure.

[0111] Figure 95 is a timing diagram illustrating a SWAP operation according to an embodiment of the present disclosure.

[0112] Figure 96 is a timing diagram illustrating a STW operation according to an embodiment of the present disclosure.

[0113] Figure 97 is a timing diagram illustrating a EXCH operation according to an embodiment of the present disclosure.

[0114] Figure 98 is a timing diagram illustrating a BSW operation according to an embodiment of the present disclosure.

[0115] Figure 99 is a timing diagram illustrating a BTSTW operation according to an embodiment of the present disclosure.

[0116] Figure 100 is a timing diagram illustrating a BCLRF, the BTSTSE, the BTSTF, the BTGF and BSETF operations according to an embodiment of the present disclosure.

[0117] Figure 101 is a timing diagram illustrating a BSET, the BTG, the BTST, the BTSTS, and the BCLR operations according to an embodiment of the present disclosure.

[0118] Figure 102 is a timing diagram illustrating a BTSS, the BTSC, the BTFSC and the BTFSS operations according to an embodiment of the present disclosure.

[0119] Figure 103 is a timing diagram illustrating a TBLRDH and the TBLRDL operations according to an embodiment of the present disclosure.

[0120] Figure 104 is a timing diagram illustrating a TBLWTH and the TBLWTL operations according to an embodiment of the present disclosure.

[0121] Figure 105 is a timing diagram illustrating a LDQW operation according to an embodiment of the present disclosure.

[0122] Figure 106 is a timing diagram illustrating a LDDW operation according to an embodiment of the present disclosure.

[0123] Figure 107 is a timing diagram illustrating a STQW operation according to an embodiment of the present disclosure.

[0124] Figure 108 is a timing diagram illustrating a STDW operation according to an embodiment of the present disclosure.

[0125] Figure 109 is a timing diagram illustrating a MULS, the MULSU, the MULSULS, the MULU, the MULULS and the MULUS operations according to an embodiment of the present disclosure.

[0126] Figure 110 is a timing diagram illustrating a MULWF operation according to an embodiment of the present disclosure.

[0127] Figure 111 is a timing diagram illustrating an ALL BRANCHES operation according to an embodiment of the present disclosure.

[0128] Figure 112 is a timing diagram illustrating a BRAW operation according to an embodiment of the present disclosure.

[0129] Figure 113 is a timing diagram illustrating a RCALL, and the RCALLW operations according to an embodiment of the present disclosure.

[0130] Figure 114 is a timing diagram illustrating a CALLW operation according to an embodiment of the present disclosure.

[0131] Figure 115 is a timing diagram illustrating a CALL operation according to an embodiment of the present disclosure.

[0132] Figure 116 is a timing diagram illustrating a GOTOW operation according to an embodiment of the present disclosure.

[0133] Figure 117 is a timing diagram illustrating a GOTO operation according to an embodiment of the present disclosure.

[0134] Figure 118 is a timing diagram illustrating a LNK operation according to an embodiment of the present disclosure.

[0135] Figure 119 is a timing diagram illustrating a ULNK operation according to an embodiment of the present disclosure.

[0136] Figure 120 is a timing diagram illustrating a DAW operation according to an embodiment of the present disclosure.

[0137] Figure 121 is a timing diagram illustrating a SCRATCH operation according to an embodiment of the present disclosure.

[0138] Figure 122 is a timing diagram illustrating a ITCH operation according to an embodiment of the present disclosure.

[0139] Figure 123 is a timing diagram illustrating a PUSH operation according to an embodiment of the present disclosure.

[0140] Figure 124 is a timing diagram illustrating a POP operation according to an embodiment of the present disclosure.

[0141] Figure 125 is a timing diagram illustrating a LDW operation according to an embodiment of the present disclosure.

[0142] Figure 126 is a timing diagram illustrating a TRAP operation according to an embodiment of the present disclosure.

[0143] Figure 127 is a timing diagram illustrating a DISI operation according to an embodiment of the present disclosure.

[0144] Figure 128 is a timing diagram illustrating a LDW operation according to an embodiment of the present disclosure.

[0145] Figure 129 is a timing diagram illustrating a DO and the DOW operations according to an embodiment of the present disclosure.

[0146] Figure 130 is a timing diagram illustrating a DO and the DOW operations continued according to an embodiment of the present disclosure.

[0147] Figure 131 is a timing diagram illustrating a MAC, the CLRAC, the EDAC, the SQRAC and the MOVSAC operations according to an embodiment of the present disclosure.

[0148] Figure 132 is a timing diagram illustrating a SQR, the ED, the MPY, the MPYN and the MSC operations according to an embodiment of the present disclosure.

[0149] Figure 133 is a timing diagram illustrating a LSRW, the LSRK, the ASRK, the ASRW, the SLW and the SLK operations according to an embodiment of the present disclosure.

[0150] Figure 134 is a timing diagram illustrating a ADDAB, the NEGAB and the SUBAB operations according to an embodiment of the present disclosure.

[0151] Figure 135 is a timing diagram illustrating an ADDAC operation according to an embodiment of the present disclosure.

[0152] Figure 136 is a timing diagram illustrating a LAC operation according to an embodiment of the present disclosure.

[0153] Figure 137 is a timing diagram illustrating a SAC and the SAC.R operations according to an embodiment of the present disclosure.

[0154] Figure 138 is a timing diagram illustrating a SETACK and the SFTAC operations according to an embodiment of the present disclosure.

[0155] Figure 139 is a timing diagram illustrating a RETURN, the RE and the TEIE operations according to an embodiment of the present disclosure.

[0156] Figure 140 is a timing diagram illustrating a MSLK, the MSRK, the MSLW and the MSRW operations according to an embodiment of the present disclosure.

[0157] Figure 141 is a timing diagram illustrating a FBCL, the FBCR, the FFOL, the FFOR, the FFIL and the FFIR operations according to an embodiment of the present disclosure.

[0158] Figure 142 is a timing diagram illustrating a RETLW operation according to an embodiment of the present disclosure.

[0159] Figure 143 is a timing diagram illustrating a REPEAT and the REPEAT W operations according to an embodiment of the present disclosure.

[0160] Figure 144 is a timing diagram illustrating a REPEAT and the REPEAT W operations continued according to an embodiment of the present disclosure.

[0161] Figure 145 is a block diagram illustrating a CPU Core according to an embodiment of the present disclosure.

[0162] Figure 146 is a block diagram illustrating data alignment according to an embodiment of the present disclosure.

[0163] Figure 147 is a block diagram illustrating a Data Space Memory Map Example according to an embodiment of the present disclosure.

[0164] Figure 148 is a block diagram illustrating a data space for a microcontroller and digital signal processor instructions according to an embodiment of the present disclosure.

[0165] Figure 149 is a block diagram illustrating a data space window into the program space operation according to an embodiment of the present disclosure.

[0166] Figure 150 is a block diagram illustrating a PS Data Read-Through DS operation according to an embodiment of the present disclosure.

[0167] Figure 151 is a block diagram illustrating a PS Data Read-Through DS within a REPEAT loop operation according to an embodiment of the present disclosure.

[0168] Figure 152 is a block diagram illustrating a data access operation from program space address generation according to an embodiment of the present disclosure.

[0169] Figure 153 is a block diagram illustrating an instruction fetch according to an embodiment of the present disclosure.

[0170] Figure 154 is a block diagram illustrating a program space memory map according to an embodiment of the present disclosure.

[0171] Figure 155 is a block diagram illustrating a program data table access according to an embodiment of the present disclosure.

[0172] Figure 156 is a block diagram illustrating program data table access according to an embodiment of the present disclosure.

[0173] Figure 157 is a block diagram illustrating HEX file compatibility according to an embodiment of the present disclosure.

[0174] Figure 158 is a basic core timing diagram according to an embodiment of the present disclosure.

[0175] Figure 159 is a timing diagram illustrating a clock/instruction cycle according to an embodiment of the present disclosure.

[0176] Figure 160 is a flow chart illustrating a REPEAT[W] loop functional flow according to an embodiment of the present disclosure.

[0177] Figure 161 is a block diagram illustrating a REPEAT[W] instruction pipeline Flow according to an embodiment of the present disclosure.

[0178] Figure 162 is a block diagram of a Do Loop hardware operation according to an embodiment of the present disclosure.

[0179] Figure 163 is timing diagram of a DO loop entry operation according to an embodiment of the present disclosure.

[0180] Figure 164 is a timing diagram illustrating a DO loop continuation operation according to an embodiment of the present disclosure.

[0181] Figure 165 is a timing diagram illustrating a DO Continue with Branch to Last Instruction operation according to an embodiment of the present disclosure.

[0182] Figure 166 is a timing diagram illustrating a DO loop EXIT operation according to an embodiment of the present disclosure.

[0183] Figure 167 is a flow chart illustrating a DO and REPEAT operation according to an embodiment of the present disclosure.

[0184] Figure 168 is a block diagram illustrating an Uninitialized W Register Trap operation according to an embodiment of the present disclosure.

[0185] Figure 169 is a block diagram illustrating a Stack Pointer Overflow & Underflow Trap operation according to an embodiment of the present disclosure.

[0186] Figure 170 is a timing diagram illustrating a Stack Timing of a PC PUSH CALL operation according to an embodiment of the present disclosure.

[0187] Figure 171 is a timing diagram illustrating a Stack Timing of a PC POP RETURN operation according to an embodiment of the present disclosure.

[0188] Figure 172 is a block diagram illustrating a CALL stack frame according to an embodiment of the present disclosure.

[0189] Figure 173a is a block diagram illustrating a stack pointer at initialization according to an embodiment of the present disclosure.

[0190] Figure 173b is a block diagram illustrating a stack pointer after a PUSH operation according to an embodiment of the present disclosure.

[0191] Figure 173c is a block diagram illustrating a stack pointer after a PUSH operation according to an embodiment of the present disclosure.

[0192] Figure 173d is a block diagram illustrating a stack pointer after a POP operation according to an embodiment of the present disclosure.

[0193] While the present invention is susceptible to various modifications and alternative forms, specific exemplary embodiments thereof have been shown by way of example in the drawings and are herein described in detail. It should be understood, however, that the description herein of specific embodiments is not intended to limit the invention to the particular forms disclosed, but on the contrary, the intention is to cover all modifications, equivalents, and alternatives falling within the spirit and scope of the invention as defined by the appended claims.

DETAILED DESCRIPTION

[0194] [0017]-In order to describe the instruction set and its relationship to a processor for executing the instruction set, an overview of pertinent processor elements is first presented with reference to FIGS-Figures 1 and 2. The overview section describes the process of fetching,

decoding and executing program instructions taken from the instruction set according to embodiments of the present invention.

~~{0018}~~ Overview of Processor Elements

{0195} ~~{0019}~~ FIG. Figure 1 depicts a functional block diagram of an embodiment of a processor chip within which the present invention may find application. Referring to FIG. Figure 1, a processor 100 is coupled to external devices/systems 140. The processor 100 may be any type of processor including, for example, a digital signal processor (DSP), a microprocessor, a microcontroller or combinations thereof. The external devices 140 may be any type of systems or devices including input/output devices such as keyboards, displays, speakers, microphones, memory, or other systems which may or may not include processors. Moreover, the processor 100 and the external devices 140 may together comprise a stand alone system.

{0196} ~~{0020}~~—The processor 100 includes a program memory 105, an instruction fetch/decode unit 110, instruction execution units 115, data memory and registers 120, peripherals 125, data I/O 130, and a program counter and loop control unit 135. The bus 150, which may include one or more common buses, communicates data between the units as shown.

{0197} ~~{0021}~~—The program memory 105 stores software embodied in program instructions for execution by the processor 100. The program memory 105 may comprise any type of nonvolatile memory such as a read only memory (ROM), a programmable read only memory (PROM), an electrically programmable or an electrically programmable and erasable read only memory (EPROM or EEPROM) or flash memory. In addition, the program memory 105 may be supplemented with external nonvolatile memory 145 as shown to increase the complexity of software available to the processor 100. Alternatively, the program memory may

be volatile memory which receives program instructions from, for example, an external non-volatile memory 145. When the program memory 105 is nonvolatile memory, the program memory may be programmed at the time of manufacturing the processor 100 or prior to or during implementation of the processor 100 within a system. In the latter scenario, the processor 100 may be programmed through a process called in-line serial programming.

[0198] ~~{0022}~~ The instruction fetch/decode unit 110 is coupled to the program memory 105, the instruction execution units 115 and the data memory 120. Coupled to the program memory 105 and the bus 150 is the program counter and loop control unit 135. The instruction fetch/decode unit 110 fetches the instructions from the program memory 105 specified by the address value contained in the program counter 135. The instruction fetch/decode unit 110 then decodes the fetched instructions and sends the decoded instructions to the appropriate execution unit 115. The instruction fetch/decode unit 110 may also send operand information including addresses of data to the data memory 120 and to functional elements that access the registers.

[0199] ~~{0023}~~ The program counter and loop control unit 135 includes a program counter register (not shown) which stores an address of the next instruction to be fetched. During normal instruction processing, the program counter register may be incremented to cause sequential instructions to be fetched. Alternatively, the program counter value may be altered by loading a new value into it via the bus 150. The new value may be derived based on decoding and executing a flow control instruction such as, for example, a branch instruction. In addition, the loop control portion of the program counter and loop control unit 135 may be used to provide repeat instruction processing and repeat loop control as further described below.

[0200] ~~{0024}~~ The instruction execution units 115 receive the decoded instructions from the instruction fetch/decode unit 110 and thereafter execute the decoded instructions. As part of

this process, the execution units may retrieve one or two operands via the bus 150 and store the result into a register or memory location within the data memory 120. The execution units may include an arithmetic logic unit (ALU) such as those typically found in a microcontroller. The execution units may also include a digital signal processing engine, a floating point processor, an integer processor or any other convenient execution unit. A preferred embodiment of the execution units and their interaction with the bus 150, which may include one or more buses, is presented in more detail below with reference to ~~FIG.~~Figure 2.

[0201] ~~{0025}~~—The data memory and registers 120 are volatile memory and are used to store data used and generated by the execution units. The data memory 120 and program memory 105 are preferably separate memories for storing data and program instructions respectively. This format is known generally as a Harvard architecture. It is noted, however, that according to the present invention, the architecture may be a Von-Neuman architecture or a modified Harvard architecture which permits the use of some program space for data space. A dotted line is shown, for example, connecting the program memory 105 to the bus 150. This path may include logic for aligning data reads from program space such as, for example, during table reads from program space to data memory 120.

[0202] ~~{0026}~~—Referring again to ~~FIG.~~Figure 1, a plurality of peripherals 125 on the processor may be coupled to the bus 125. The peripherals may include, for example, analog to digital converters, timers, bus interfaces and protocols such as, for example, the controller area network (CAN) protocol or the Universal Serial Bus (USB) protocol and other peripherals. The peripherals exchange data over the bus 150 with the other units.

[0203] ~~{0027}~~—The data I/O unit 130 may include transceivers and other logic for interfacing with the external devices/systems 140. The data I/O unit 130 may further include

functionality to permit in circuit serial programming of the Program memory through the data I/O unit 130.

[0204] ~~[0028]~~ ~~FIG.~~Figure 2 depicts a functional block diagram of a data busing scheme for use in a processor 100, such as that shown in ~~FIG.~~Figure 1, which has an integrated microcontroller arithmetic logic unit (ALU) 270 and a digital signal processing (DSP) engine 230. This configuration may be used to integrate DSP functionality to an existing microcontroller core. Referring to ~~FIG.~~Figure 2, the data memory 120 of ~~FIG.~~Figure 1 is implemented as two separate memories: an X-memory 210 and a Y-memory 220, each being respectively addressable by an X-address generator 250 and a Y-address generator 260. The X-address generator may also permit addressing the Y-memory space thus making the data space appear like a single contiguous memory space when addressed from the X address generator. The bus 150 may be implemented as two buses, one for each of the X and Y memory, to permit simultaneous fetching of data from the X and Y memories.

[0205] ~~[0029]~~ The W registers 240 are general purpose address and/or data registers. The DSP engine 230 is coupled to both the X and Y memory buses and to the W registers 240. The DSP engine 230 may simultaneously fetch data from each the X and Y memory, execute instructions which operate on the simultaneously fetched data and write the result to an accumulator (not shown) and write a prior result to X or Y memory or to the W registers 240 within a single processor cycle.

[0206] ~~[0030]~~ In one embodiment, the ALU 270 may be coupled only to the X memory bus and may only fetch data from the X bus. However, the X and Y memories 210 and 220 may be addressed as a single memory space by the X address generator in order to make the data

memory segregation transparent to the ALU 270. The memory locations within the X and Y memories may be addressed by values stored in the W registers 240.

[0207] ~~{0031}~~ Any processor clocking scheme may be implemented for fetching and executing instructions. A specific example follows, however, to illustrate an embodiment of the present invention. Each instruction cycle is comprised of four Q clock cycles Q1-Q4. The four phase Q cycles provide timing signals to coordinate the decode, read, process data and write data portions of each instruction cycle.

[0208] ~~{0032}~~ According to one embodiment of the processor 100, the processor 100 concurrently performs two operations--it fetches the next instruction and executes the present instruction. Accordingly, the two processes occur simultaneously. The following sequence of events may comprise, for example, the fetch instruction cycle:

- 1-Q1: Fetch Instruction
- Q2: Fetch Instruction
- Q3: Fetch Instruction
- Q4: Latch Instruction into prefetch register, Increment PC

[0209] ~~{0033}~~ The following sequence of events may comprise, for example, the execute instruction cycle for a single operand instruction:

- 2-Q1: latch instruction into IR, decode and determine addresses of operand data
- Q2: fetch operand
- Q3: execute function specified by instruction and calculate destination address for data
- Q4: write result to destination

[0210] ~~{0034}~~ The following sequence of events may comprise, for example, the execute instruction cycle for a dual operand instruction using a data pre-fetch mechanism. These instructions pre-fetch the dual operands simultaneously from the X and Y data memories and store them into registers specified in the instruction. They simultaneously allow instruction execution on the operands fetched during the previous cycle.

- 3-Q1: latch instruction into IR, decode and determine addresses of operand data
- Q2: pre-fetch operands into specified registers, execute operation in instruction
- Q3: execute operation in instruction, calculate destination address for data
- Q4: complete execution, write result to destination

~~[0035]~~ DSP Digital Signal Processing Engine

[0211] ~~[0036]~~ FIG. Figure 3 depicts a functional block diagram of the DSP engine 230.

The DSP engine executes various instructions within the instruction set according to embodiments of the present invention. The DSP engine 230 is coupled to the X and the Y bus and the W registers 240. The DSP engine includes a multiplier 300, a barrel shifter 330, an adder/subtractor 340, two accumulators 345 and 350 and round and saturation logic 365. These elements and others that are discussed below with reference to ~~FIG.~~ Figure 3 cooperate to process DSP instructions including, for example, multiply and accumulate instructions and shift instructions. According to one embodiment of the invention, the DSP engine operates as an asynchronous block with only the accumulators and the barrel shifter result registers being clocked. Other configurations, including pipelined configurations, may be implemented according to the present invention.

[0212] ~~[0037]~~ The multiplier 300 has inputs coupled to the W registers 240 and an output coupled to the input of a multiplexer 305. The multiplier 300 may also have inputs coupled to the X and Y bus. The multiplier may be any size however, for convenience, a ~~16-times.~~ 16x16 bit multiplier is described herein which produces a 32 bit output result. The multiplier may be capable of signed and unsigned operation and can multiplex its output using a scaler to support either fractional or integer results.

[0213] ~~[0038]~~ The output of the multiplier 300 is coupled to one input of a multiplexer 305. The multiplexer 305 has another input coupled to zero backfill logic 310, which is coupled to the X Bus. The zero backfill logic 310 is included to illustrate that 16 zeros may be

concatenated onto the 16 bit data read from the X bus to produce a 32 bit result fed into the multiplexer 305. The 16 zeros are generally concatenated into the least significant bit positions.

[0214] ~~[0039]~~ The multiplexer 305 includes a control signal controlled by the instruction decoder of the processor which determines which input, either the multiplier output or a value from the X bus is passed forward. For instructions such as multiply and accumulate (MAC), the output of the multiplier is selected. For other instructions such as shift instructions, the value from the X bus (via the zero backfill logic) may be selected. The output of the multiplexer 305 is fed into the sign extend unit 315.

[0215] ~~[0040]~~ The sign extend unit 315 sign extends the output of the multiplexer from a 32 bit value to a 40 bit value. The sign extend unit 315 is illustrative only and this function may be implemented in a variety of ways. The sign extend unit 315 outputs a 40 bit value to a multiplexer 320.

[0216] ~~[0041]~~ The multiplexer 320 receives inputs from the sign extend unit 315 and the accumulators 345 and 350. The multiplexer 320 selectively outputs values to the input of a barrel shifter 330 based on control signals derived from the decoded instruction. The accumulators 345 and 350 may be any length. According to the embodiment of the present invention selected for illustration, the accumulators are 40 bits in length. A multiplexer 360 determines which accumulator 345 or 350 is output to the multiplexer 320 and to the input of an adder 340.

[0217] ~~[0042]~~ The instruction decoder sends control signals to the multiplexers 320 and 360, based on the decoded instruction. The control signals determine which accumulator is selected for either an add operation or a shift operation and whether a value from the multiplier or the X bus is selected for an add operation or a shift operation.

[0218] ~~[0043]~~ The barrel shifter 330 performs shift operations on values received via the multiplexer 320. The barrel shifter may perform arithmetic and logical left and right shifts and circular shifts where bits rotated out one side of the shifter reenter through the opposite side of the buffer. In the illustrated embodiment, the barrel shifter is 40 bits in length and may perform a 15 bit arithmetic right shift and a 16 bit left shift in a single cycle. The shifter uses a signed binary value to determine both the magnitude and the direction of the shift operation. The signed binary value may come from a decoded instruction, such as shift instruction or a multi-precision shift instruction. According to one embodiment of the invention, a positive signed binary value produces a right shift and a negative signed binary value produces a left shift.

[0219] ~~[0044]~~ The output of the barrel shifter 330 is sent to the multiplexer 355 and the multiplexer 370. The multiplexer 355 also receives inputs from the accumulators 345 and 350. The multiplexer 355 operates under control of the instruction decoder to selectively apply the value from one of the accumulators or the barrel shifter to the adder/subtractor 340 and the round and saturate logic 365.

[0220] ~~[0045]~~ The adder/subtractor 340 may select either accumulator 345 or 350 as a source and/or a destination. In the illustrated embodiment, the adder/subtractor 340 has 40 bits. The adder receives an accumulator input and an input from another source such as the barrel shifter 331, the X bus or the multiplier. The value from the barrel shifter 331 may come from the multiplier or the X bus and may be scaled in the barrel shifter prior to its arrival at the other input of the adder/subtractor 340. The adder/subtractor 340 adds to or subtracts a value from the accumulator and stores the result back into one of the accumulators. In this manner values in the accumulators represent the accumulation of results from a series of arithmetic operations. The round and saturate logic 365 is used to round 40 bit values from the accumulator or the barrel

shifter down to 16 bit values that may be transmitted over the X bus for storage into a W register or data memory. The round and saturate logic has an output coupled to a multiplexer 370. The multiplier 370 may be used to select either the output of the round and saturate logic 365 or the output from a selected 16 bits of the barrel shifter 330 for output to the X bus.

{0046} Description of the Instruction Set

{0221} ~~{0047}~~ The designated instruction set according to the present invention is set forth in Table 1-1, which lists the instruction set the following tables, and are listed in alphabetical order using mnemonics. The mnemonics are merely illustrative, and one of ordinary skill in the art will understand that alternate mnemonics may be used to achieve the same result. The designated instruction set and descriptions of each designated instruction is presented in Appendix A. All of the tables are set forth at the end of the specification prior to the Figures. There are ninety four instructions, many of which have several addressing modes. the following tables. To simplify the definition, each variant of an instruction is given a different "PLA mnemonic." The detailed definitions of the instructions are listed by the PLA mnemonic in each table Table 1-1 which lists the illustrative assembly syntax of each mnemonic, gives examples of usage of that syntax, gives the PLA mnemonic and references an appendix page at which a description of the instruction is found. Symbols used in the definitions of Table 1-1 the tables of the instruction set are defined in Table 6-1 found in Appendix A. Appendix A comprises additional details describing the operation of each instruction and is incorporated by reference herein. 1.

TABLE 1 -- Symbols used in the Opcode Descriptions

<u>Field</u>	<u>Description</u>
<u>{ }</u>	<u>Optional field or operation</u>
<u>[text]</u>	<u>Means "the location addressed by text"</u>
<u>(text)</u>	<u>Means "content of text"</u>
<u>#text</u>	<u>Means literal defined by "text"</u>
<u>text1 ∈ {text2, text3, ...}</u>	<u>text1 must be in the set of text2, text3, ...</u>

<u>none</u>	<u>field does not require an entry, may be blank</u>
<u>{label:}</u>	<u>Optional Label name</u>
<u>label</u>	<u>Translates to a literal representing the location of the label name</u>
<u><n:m></u>	<u>Register bit field</u>
<u>lit1</u>	<u>1-bit unsigned literal $\in \{0,1\}$</u>
<u>lit4</u>	<u>4-bit unsigned literal $\in \{0...15\}$</u>
<u>lit5</u>	<u>5-bit unsigned literal $\in \{0...31\}$</u>
<u>Slit5</u>	<u>5-bit signed literal $\in \{-16...15\}$</u>
<u>Slit10</u>	<u>10-bit signed literal $\in \{-512...511\}$</u>
<u>lit14</u>	<u>14-bit unsigned literal $\in \{0...16384\}$</u>
<u>lit16</u>	<u>16-bit unsigned literal $\in \{0...65535\}$</u>
<u>Slit16</u>	<u>16-bit signed literal $\in \{-32768...32767\}$</u>
<u>lit23</u>	<u>23-bit unsigned literal $\in \{0...8388608\}$; LSB must be 0</u>
<u>bit3</u>	<u>3-bit bit selection field (used in byte addressed instructions) $\in \{0...7\}$</u>
<u>bit4</u>	<u>4-bit bit selection field (used in word addressed instructions) $\in \{0...15\}$</u>
<u>.w</u>	<u>Word mode selection (default)</u>
<u>.b</u>	<u>Byte mode selection</u>
<u>.s</u>	<u>Shadow register select</u>
<u>f</u>	<u>File register address $\in \{0000h...1FFFh\}$</u>
<u>d</u>	<u>File register destination $d \in \{Ww, none\}$</u>
<u>Ww</u>	<u>Default W working register (used in file register instructions)</u>
<u>Wn</u>	<u>One of 16 working registers $\in \{W0..W15\}$</u>
<u>Wns</u>	<u>One of 16 source working registers $\in \{W0..W15\}$</u>
<u>Wnd</u>	<u>One of 16 destination working registers $\in \{W0..W15\}$</u>
<u>Wb</u>	<u>Base W register $\in \{W0..W15\}$</u>
<u>Ws</u>	<u>Source W register $\in \{Ws, [Ws], [Ws]++, [Ws]--, [Ws++]\}$</u>
<u>Wd</u>	<u>Destination W register $\in \{Wd, [Wd], [Wd]++, [Wd]--, [Wd++]\}$</u>
<u>Wso</u>	<u>Source W register $\in \{Wns, [Wns], [Wns]++, [Wns]--, [Wns--], [Wns+Wb], [Wns+slit5]\}$</u>
<u>Wdo</u>	<u>Destination W register $\in \{Wnd, [Wnd], [Wnd]++, [Wnd]--, [Wnd--], [Wnd+Wb], [Wnd+slit5]\}$</u>
<u>Wm*Wm</u>	<u>Multiplicand and Multiplier W register for Square instructions $\in \{W0*W0, W1*W1, W2*W2, W3*W3\}$</u>
<u>Wm*Wn</u>	<u>Multiplicand and Multiplier W register for DSP instructions $\in \{W0*W1, W0*W2, W0*W3, W1*W2, W1*W3, W2*W3\}$</u>
<u>Wx</u>	<u>X data space prefetch address register for DSP instructions $\in \{[W4] += 6, [W4] += 4, [W4] += 2, [W4], [W4] -= 6, [W4] -= 4, [W4] -= 2, [W5] += 6, [W5] += 4, [W5] += 2, [W5], [W5] -= 6, [W5] -= 4, [W5] -= 2, [W5+W8], none\}$</u>
<u>Wy</u>	<u>Y data space prefetch address register for DSP instructions $\in \{[W6] += 8, [W6] += 4, [W6] += 2, [W6], [W6] -= 6, [W6] -= 4, [W6] -= 2, [W7] += 8, [W7] += 4, [W7] += 2, [W7], [W7] -= 6, [W7] -= 4, [W7] -= 2, [W7+W8], none\}$</u>
<u>Wxp</u>	<u>X data space prefetch destination register for DSP instructions $\in \{W0..W3\}$</u>
<u>Wyp</u>	<u>Y data space prefetch destination register for DSP instructions $\in \{W0..W3\}$</u>
<u>AWB</u>	<u>Accumulator write back destination address register $\in \{W9, [W9]++\}$</u>
<u>PC</u>	<u>Program Counter</u>
<u>PCL</u>	<u>Program Counter Low Byte</u>
<u>PCH</u>	<u>Program Counter High Byte</u>
<u>PCU</u>	<u>Program Counter Upper Byte</u>
<u>PCLATH</u>	<u>Program Counter High Byte Latch</u>

PCLATU

OA, OB, SA, SB

C, DC, N, OV, SZ, Z

Program Counter Upper Byte Latch

DSC status bits: ACCA Overflow, ACCB Overflow, ACCA Saturate, ACCB Saturate

ALU status bits: Carry, Digit Carry, Negative, Overflow, Sticky-Zero, Zero

TABLE 2 -- ADD: Add Wb and Ws

Syntax:	<u>{label:}</u>	<u>ADD{.b}</u>	<u>Wb,</u>	<u>Ws,</u>	<u>Wd</u>	
				<u>[Ws],</u>	<u>[Wd]</u>	
				<u>[Ws]++,</u>	<u>[Wd]++</u>	
				<u>[Ws]--,</u>	<u>[Wd]--</u>	
				<u>[Ws++]</u> ,	<u>[Wd++]</u>	
				<u>[Ws--]</u> ,	<u>[Wd--]</u>	
Operands:	<u>Wb ∈ [W0 ... W15]; Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]</u>					
Operation:	<u>(Wb) + (Ws) → Wd</u>					
Status Affected:	<u>C, DC, N, OV, Z</u>					
Encoding:	<u>0100</u>	<u>0www</u>	<u>wBqq</u>	<u>qddd</u>	<u>dppp</u>	<u>ssss</u>
Description:	<u>Add the contents of the source register Ws and the contents of the base register Wb and place the result in the destination register Wd.</u> <u>The ‘B’ bit selects byte or word operation.</u> <u>The ‘s’ bits select the address of the source register.</u> <u>The ‘w’ bits select the address of the base register.</u> <u>The ‘d’ bits select the address of the destination register.</u> <u>The ‘p’ bits select source address mode 2.</u> <u>The ‘q’ bits select destination address mode 2.</u> <u>See Table 1-5 and Table 1-6 for modifier addressing information.</u> <u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>ADD</u>	<u>W5,W6,W7</u>		<u>; Add</u>	
	<u>Before Instruction</u> <u>After Instruction</u>					

TABLE 3 -- ADDAB: Add ACCA to ACCB

<u>Syntax:</u>	<u>{label:}</u>	<u>ADD</u>	<u>A</u>			
			<u>B</u>			
<u>Operands:</u>	<u>none</u>					
<u>Operation:</u>	<u>(ACCA) + ACCB → ACC(A or B)</u>					
<u>Status Affected:</u>	<u>OA, OB, SA, SB</u>					
<u>Encoding:</u>	<u>1100</u>	<u>1011</u>	<u>A000</u>	<u>0000</u>	<u>0000</u>	<u>0000</u>
<u>Description:</u>	<u>Add ACCA to ACCB and write results to selected accumulator.</u>					
	<u>The 'A' bits specify the destination accumulator.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1</u>					
<u>Example:</u>		<u>ADD</u>	<u>B</u>	<u>; Add ACCA to ACCB, result to ACCB</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 4 -- ADDAC: 16-Bit Signed Add to Accumulator

Syntax:	{label:}	ADD	A,	Wns,	[, Slit4]	
			B,	[Wns],		
				[Wns]++		
				[Wns]--		
				[Wns--],		
				[Wns+Wb],		
				[Wns+lit5]		
Operands:	Wns \in [W0 ... W15]; Wb \in [W0 ... W15]; lit5 \in [0 ... 31] Slit4 \in [-8 ... +7]					
Operation:	(ACC) + Shift_{Slit4}(Extend(Wns)) \rightarrow ACC					
Status Affected:	OA, OB, SA, SB					
Encoding:	1100	1001	Awww	wrrr	rggg	ssss
Description:	<p>The term contained at the effective address is assumed to be Q15 fractional data and is automatically sign-extended and zero-backfilled prior to the operation.</p> <p>Optionally shift the term, then add the term to accumulator.</p> <p>The 'A' bits specify the destination accumulator.</p> <p>The 's' bits specify the source register Wns.</p> <p>The 'g' bits select source address mode 3.</p> <p>The 'w' bits specify the offset amount lit5 OR the offset register Wb.</p> <p>The 'r' bits encode the optional operand Slit4 which determines the amount of the accumulator preshift; if the operand Slit4 is absent, a 0 is encoded.</p> <p>See Table 1-7 for modifier addressing information.</p> <p>Note: Positive values of operand Slit4 represent arithmetic shift right. Negative values of operand Slit4 represent shift left.</p>					
Words:	1					
Cycles:	1					
Example:	ADD	A,W5,# 3	; Shift W5 right 3 bits, add to accumulator A			
	Before Instruction					
	After Instruction					

TABLE 5 -- ADDC: Add Wb and Ws with Carry

Syntax:	{label:}	ADDC{.b}	Wb,	Ws,	Wd	
				[Ws],	[Wd]	
				[Ws]++,	[Wd]++	
				[Ws]--,	[Wd]--	
				[Ws++] ,	[Wd++]	
				[Ws--],	[Wd--]	
Operands:	Wb ∈ [W0 ... W15]; Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]					
Operation:	(Wb) + (Ws) + (C) → Wd					
Status Affected:	C, DC, N, OV, Z					
Encoding:	0100	1www	wBqq	qddd	dppp	Ssss
Description:	<p>Add the contents of the source register Ws and the contents of the base register Wb and the Carry bit and place the result in the destination register Wd.</p> <p>The 'B' bit selects byte or word operation.</p> <p>The 's' bits select the address of the source register.</p> <p>The 'w' bits select the address of the base register.</p> <p>The 'd' bits select the address of the destination register.</p> <p>The 'p' bits select source address mode 2.</p> <p>The 'q' bits select destination address mode 2.</p> <p>See Table 1-5 and Table 1-6 for modifier addressing information.</p> <p>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</p>					
Words:	1					
Cycles:	1					
Example:		ADDC		W5,W6,W7	: Add	
	Before Instruction					
	After Instruction					

TABLE 6 -- ADDCLS: Add Wb and Short Literal with Carry

Syntax:	{label:}	ADDC{.b}	Wb,	lit5,	Wd	
					[Wd]	
					[Wd]++	
					[Wd]--	
					[Wd++]	
					[Wd--]	
Operands:	Wb ∈ [W0 ... W15]; lit5 ∈ [0 ... 31]; Wd ∈ [W0 ... W15]					
Operation:	(Wb) + lit5 + (C) → Wd					
Status Affected:	C, DC, N, OV, Z					
Encoding:	0100	1www	wBgg	gddd	d11k	kkkk
Description:	<p>Add the contents of the base register Wb, the literal operand and the Carry bit; and place the result in the destination register Wd.</p> <p>The 'B' bit selects byte or word operation.</p> <p>The 'w' bits select the address of the base register.</p> <p>The 'k' bits provide the literal operand, a five-bit integer number.</p> <p>The 'd' bits select the address of the destination register.</p> <p>The 'q' bits select destination address mode 2.</p> <p>See Table 1-6 for modifier addressing information.</p> <p>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</p>					
Words:	1					
Cycles:	1					
Example:		ADDC		W5,#12,W7	; Add	
	Before Instruction					
	After Instruction					

TABLE 7 -- ADDCLW: Add Literal to Wn with Carry

Syntax:	{label:}	ADDC{.b}	Slit10,	Wn		
Operands:	<u>Slit10</u> ∈ [-512 ... 511]; <u>Wn</u> ∈ [W0 ... W15]					
Operation:	<u>Slit10 + (Wn) + (C) → Wn</u>					
Status Affected:	<u>C, DC, N, OV, Z</u>					
Encoding:	<u>1011</u>	<u>0000</u>	<u>1Bkk</u>	<u>kkkk</u>	<u>kkkk</u>	<u>dddd</u>
Description:	<p><u>Add the literal operand to the contents of the working register Wn and the Carry bit and place the result in the working register Wn.</u></p> <p><u>The 'B' bit selects byte or word operation.</u></p> <p><u>The 'd' bits select the address of the working register.</u></p> <p><u>The 'k' bits specify the literal operand, a signed 10-bit number.</u></p> <p><u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>ADDC</u>	<u>#123,W7</u>	<u>: Add w/ carry</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 8 -- ADDLS: Add Wb and Short Literal

Syntax:	<u>{label:}</u>	<u>ADD{.b}</u>	<u>Wb,</u>	<u>lit5,</u>	<u>Wd</u>	
					<u>[Wd]</u>	
					<u>[Wd]++</u>	
					<u>[Wd]--</u>	
					<u>[Wd++]</u>	
					<u>[Wd--]</u>	
Operands:	<u>Wb ∈ [W0 ... W15]; lit5 ∈ [0 ... 31]; Wd ∈ [W0 ... W15]</u>					
Operation:	<u>(Wb) + lit5 → Wd</u>					
Status Affected:	<u>C, DC, N, OV, Z</u>					
Encoding:	<u>0100</u>	<u>0www</u>	<u>wBqq</u>	<u>qddd</u>	<u>d11k</u>	<u>kkkk</u>
Description:	<p><u>Add the contents of the source register Ws and the literal operand and place the result in the destination register Wd.</u></p> <p><u>The 'B' bit selects byte or word operation.</u></p> <p><u>The 'w' bits select the address of the base register.</u></p> <p><u>The 'k' bits provide the literal operand, a five-bit integer number.</u></p> <p><u>The 'd' bits select the address of the destination register.</u></p> <p><u>The 'q' bits select destination address mode 2.</u></p> <p><u>See Table 1-6 for modifier addressing information.</u></p> <p><u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>ADD</u>		<u>W5,#12,W7</u>	<u>; Add</u>	
	<u>Before Instruction</u> <u>After Instruction</u>					

TABLE 9 -- ADDLW: Add Literal to Wn

Syntax:	<u>{label:}</u>	<u>ADD{.b}</u>	<u>Slit10,</u>	<u>Wn</u>		
Operands:	<u>Slit10</u> \in [-512 ... 511]; <u>Wn</u> \in [W0 ... W15]					
Operation:	<u>Slit10 + (Wn) \rightarrow Wn</u>					
Status Affected:	<u>C, DC, N, OV, Z</u>					
Encoding:	<u>1011</u>	<u>0000</u>	<u>0Bkk</u>	<u>kkkk</u>	<u>kkkk</u>	<u>dddd</u>
Description:	<p><u>Add the literal operand to the contents of the working register Wn and place the result in the working register Wn.</u></p> <p><u>The 'B' bit selects byte or word operation.</u></p> <p><u>The 'd' bits select the address of the working register.</u></p> <p><u>The 'k' bits specify the literal operand, a signed 10-bit number.</u></p> <p><u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>ADD</u>	<u>#123,W7</u>	<u>:</u>	<u>Add</u>	
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 10 -- ADDWF: Add f and Ww

Syntax:	<u>{label:}</u>	<u>ADD{.b}</u>	<u>f</u>	<u>{,Ww}</u>		
Operands:	<u>f ∈ [0 ... 8191]</u>					
Operation:	<u>(f) + (Ww) → destination designated by D</u>					
Status Affected:	<u>C, DC, N, OV, Z</u>					
Encoding:	<u>1011</u>	<u>0100</u>	<u>0BDf</u>	<u>ffff</u>	<u>ffff</u>	<u>ffff</u>
Description:	<p><u>Add the contents of the working register and the contents of the file register and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.</u></p> <p><u>The 'B' bit selects byte or word operation.</u> <u>The 'D' bit selects the destination.</u> <u>The 'f' bits select the address of the file register.</u></p> <p><u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>ADD</u>	<u>RAM135, Ww</u>	<u>: Add</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 11 -- ADDWFC: Add f and Carry bit and Ww

Syntax:	{label:}	ADDC{.b}	f	{Ww}		
Operands:	$f \in [0 \dots 8191]$					
Operation:	$(f) + (Ww) + (C) \rightarrow \text{destination designated by D}$					
Status Affected:	C, DC, N, OV, Z					
Encoding:	<u>1011</u>	<u>0100</u>	<u>1BDf</u>	<u>ffff</u>	<u>ffff</u>	<u>ffff</u>
Description:	<p><u>Add the contents of the working register and the carry flag and the contents of the file register and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.</u></p> <p><u>The 'B' bit selects byte or word operation.</u></p> <p><u>The 'D' bit selects the destination.</u></p> <p><u>The 'f' bits select the address of the file register.</u></p> <p><u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>ADDC</u>	<u>RAM135, Ww</u>	<u>; Add</u>		
	<u>Before Instruction</u> <u>After Instruction</u>					

TABLE 12 -- AND: And Wb and Ws

Syntax:	<u>{label:}</u>	<u>AND{.b}</u>	<u>Wb,</u>	<u>Ws,</u>	<u>Wd</u>	
				<u>[Ws],</u>	<u>[Wd]</u>	
				<u>[Ws]++,</u>	<u>[Wd]++</u>	
				<u>[Ws]--,</u>	<u>[Wd]--</u>	
				<u>[Ws++]</u> ,	<u>[Wd++]</u>	
				<u>[Ws--]</u> ,	<u>[Wd--]</u>	
Operands:	<u>Wb ∈ [W0 ... W15]; Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]</u>					
Operation:	<u>(Wb).AND.(Ws) → Wd</u>					
Status Affected:	<u>N, Z</u>					
Encoding:	<u>0110</u>	<u>0www</u>	<u>wBqq</u>	<u>qddd</u>	<u>dppp</u>	<u>ssss</u>
Description:	<p><u>Compute the AND of the contents of the source register Ws and the contents of the base register Wb and place the result in the destination register Wd.</u></p> <p><u>The 'B' bit selects byte or word operation.</u></p> <p><u>The 's' bits select the address of the source register.</u></p> <p><u>The 'w' bits select the address of the base register.</u></p> <p><u>The 'd' bits select the address of the destination register.</u></p> <p><u>The 'p' bits select source address mode 2.</u></p> <p><u>The 'q' bits select destination address mode 2.</u></p> <p><u>See Table 1-5 and Table 1-6 for modifier addressing information.</u></p> <p><u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>AND</u>	<u>W5,W6,W7</u>	<u>:</u>	<u>And</u>	
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 13 -- ANDLS; AND Wb and Short Literal

Syntax:	<u>{label:}</u>	<u>AND{.b}</u>	<u>Wb,</u>	<u>lit5,</u>	<u>Wd</u>	
					<u>[Wd]</u>	
					<u>[Wd]++</u>	
					<u>[Wd]--</u>	
					<u>[Wd++]</u>	
					<u>[Wd--]</u>	
Operands:	<u>Wb ∈ [W0 ... W15]; lit5 ∈ [0 ... 31]; Wd ∈ [W0 ... W15]</u>					
Operation:	<u>(Wb).AND.lit5 → Wd</u>					
Status Affected:	<u>N, Z</u>					
Encoding:	<u>0110</u>	<u>0www</u>	<u>wBgg</u>	<u>qddd</u>	<u>d11k</u>	<u>kkkk</u>
Description:	<p><u>Compute the AND of the contents of the base register Wb and the literal operand and place the result in the destination register Wd.</u></p> <p><u>The 'B' bit selects byte or word operation.</u></p> <p><u>The 'w' bits select the address of the base register.</u></p> <p><u>The 'k' bits provide the literal operand, a five-bit integer number.</u></p> <p><u>The 'd' bits select the address of the destination register.</u></p> <p><u>The 'q' bits select destination address mode 2.</u></p> <p><u>See Table 1-6 for modifier addressing information.</u></p> <p><u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>AND</u>		<u>W5,#12,W7</u>	<u>: AND</u>	
	<u>Before Instruction</u> <u>After Instruction</u>					

TABLE 14 -- ANDLW: AND Literal and Wd

Syntax:	{label:}	AND{.b}	Slit10,	Wn		
Operands:	Slit10 \in [-512 ... 511]; Wn \in [W0 ... W15]					
Operation:	Slit10.AND.(Wn) \rightarrow Wn					
Status Affected:	N, Z					
Encoding:	1011	0010	0Bkk	kkkk	kkkk	dddd
Description:	<p>Compute the AND of the literal operand and the contents of the working register Wn and place the result in the working register Wn.</p> <p>The 'B' bit selects byte or word operation.</p> <p>The 'd' bits select the address of the working register.</p> <p>The 'k' bits specify the literal operand, a signed 10-bit number.</p> <p>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</p>					
Words:	1					
Cycles:	1					
Example:		AND	#123,W7	:	AND	
	Before Instruction					
	After Instruction					

TABLE 15 -- ANDWF: And f and Ww

Syntax:	{label:}	AND{.b}	f	{.Ww}		
Operands:	f ∈ [0 ... 8191]					
Operation:	(f).AND.(Ww) → destination designated by D					
Status Affected:	N, Z					
Encoding:	1011	0110	0BDf	ffff	ffff	ffff
Description:	<p>Compute the AND of the contents of the working register and the contents of the file register and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.</p> <p>The 'B' bit selects byte or word operation.</p> <p>The 'D' bit selects the destination.</p> <p>The 'f' bits select the address of the file register.</p> <p>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</p>					
Words:	1					
Cycles:	1					
Example:		AND		RAM135, Ww	; And	
	Before Instruction					
	After Instruction					

TABLE 16 -- ASR: Arithmetic Shift Right Ws

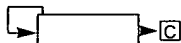
Syntax:	{label:}	ASR{.b}	Ws,	Wd		
			[Ws],	[Wd]		
			[Ws]++,	[Wd]++		
			[Ws]--,	[Wd]--		
			[Ws++]	[Wd++]		
			[Ws--]	[Wd--]		
Operands:	Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]					
Operation:	<p>For word operation: (Ws<15>) → Wd<15>, (Ws<15>) → Wd<14>, (Ws<14:1>) → Wd<13:0>, (Ws<0>) → C</p> <p>For byte operation: (Ws<7>) → Wd<7>, (Ws<7>) → Wd<6>, (Ws<6:1>) → Wd<5:0>, (Ws<0>) → C</p> 					
Status Affected:	C, N, OV, Z					
Encoding:	1101	0001	1Bqq	qddd	dppp	ssss
Description:	<p>Shift the contents of the source register Ws one bit to the right and place the result in the destination register Wd. Shift the MSB back into itself. The Carry Flag is set if the LSB of Ws is '1'.</p> <p>The 'B' bit selects byte or word operation.</p> <p>The 's' bits select the address of the source register.</p> <p>The 'd' bits select the address of the destination register.</p> <p>The 'p' bits select source address mode 2.</p> <p>The 'q' bits select destination address mode 2.</p> <p>See Table 1-5 and Table 1-6 for modifier addressing information.</p> <p>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</p>					
Words:	1					
Cycles:	1					
Example:		ASR	W5,W6	: Arithmetic shift right		
	Before Instruction					
	After Instruction					

TABLE 17 -- ASRF: Arithmetic Shift Right f

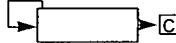
<u>Syntax:</u>	<u>{label:}</u>	<u>ASR{.b}</u>	<u>f</u>	<u>{,Ww}</u>		
<u>Operands:</u>	<u>f ∈ [0 ... 8191]</u>					
<u>Operation:</u>	<u>For word operation:</u> <u>(f<15>) → Dest<15>, (f<15>) → Dest<14></u> <u>(f<14:1>) → Dest<13:0>, (f<0>) → C</u> <u>For byte operation:</u> <u>(f<7>) → Dest<7>, (f<7>) → Dest<6>,</u> <u>(f<6:1>) → Dest<5:0>, (f<0>) → C</u> 					
<u>Status Affected:</u>	<u>C, N, OV, Z</u>					
<u>Encoding:</u>	<u>1101</u>	<u>0101</u>	<u>1BDf</u>	<u>ffff</u>	<u>ffff</u>	<u>ffff</u>
<u>Description:</u>	<u>Shift the contents of the file register f one bit to the right through the carry flag and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.</u> <u>The 'B' bit selects byte or word operation.</u> <u>The 'D' bit selects the destination.</u> <u>The 'f' bits select the address of the file register.</u> <u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1</u>					
<u>Example:</u>		<u>ASR</u>	<u>RAM135, Ww</u>	<u>; Arithmetic shift right</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 18 -- ASRK: Arithmetic Shift Right by Short Literal

Syntax:	<u>{label:}</u>	<u>ASR</u>	<u>Wb,</u>	<u>lit5,</u>	<u>Wnd</u>	
Operands:	<u>Wb</u> ∈ [W0 ... W15]; <u>lit5</u> ∈ [0...31]; <u>Wnd</u> ∈ [W0 ... W15]					
Operation:	<u>lit5<3:0>→Shift_Val</u> <u>0→Shift_In<39:32></u> <u>Wb<15:0>→Shift_In<31:16></u> <u>0→Shift_In<15:0></u> <u>0→Shift_Out<39:32></u> <u>Shift_In<31>→Shift_Out<32:32-Shift_Val></u> <u>Shift_In<31:Shift_Val>→Shift_Out<31-Shift_Val:0></u> <u>If lit5<4>==0: (less than 16)</u> <u>Shift_Out<31:16>→Wnd</u> <u>Shift_Out<15:0>→CARRY1</u> <u>0→CARRY0</u> <u>If lit5<4>==1: (16 or greater)</u> <u>Shift_Out<31:31>→Wnd<15:0></u> <u>Shift_Out<31:16>→CARRY1</u> <u>Shift_Out<15:0>→CARRY0</u>					
Status Affected:	<u>C,SZ,Z</u>					
Encoding:	<u>1101</u>	<u>1110</u>	<u>1www</u>	<u>wddd</u>	<u>d11k</u>	<u>kkkk</u>
Description:	<p><u>Arithmetic shift right the contents of the source register Wb by lit5 bits (up to 31 positions), placing the result in the destination register Wnd. Bits that are shifted beyond the rightmost position of the source are stored in the CARRY1 and CARRY0 registers.</u></p> <p><u>The Z and SZ bits will be set if the value placed in Wnd is zero and cleared otherwise. The C bit will be set if any of the bits shifted out were set (in other words, if the resultant CARRY is non-zero) and cleared otherwise.</u></p> <p><u>Note: This instruction operates in word mode only.</u></p>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					

TABLE 19 -- ASRW: Arithmetic Shift Right by Wns

Syntax:	<u>{label:}</u>	<u>ASR</u>	<u>Wb,</u>	<u>Wns,</u>	<u>Wnd</u>	
Operands:	<u>Wb ∈ [W0 ... W15]; Wns ∈ [W0 ... W15]; Wnd ∈ [W0 ... W15]</u>					
Operation:	<u>Wns<3:0>→Shift_Val</u> <u>0→Shift_In<39:32></u> <u>Wb<15:0>→Shift_In<31:16></u> <u>0→Shift_In<15:0></u> <u>0→Shift_Out<39:32></u> <u>Shift_In<31>→Shift_Out<32:32-Shift_Val></u> <u>Shift_In<31:Shift_Val>→Shift_Out<31-Shift_Val:0></u> <u>If Wns<4>==0: (less than 16)</u> <u>Shift_Out<31:16>→Wnd</u> <u>Shift_Out<15:0>→CARRY1</u> <u>0→CARRY0</u> <u>If Wns<4>==1: (16 or greater)</u> <u>Shift_Out<31:31>→Wnd<15:0></u> <u>Shift_Out<31:16>→CARRY1</u> <u>Shift_Out<15:0>→CARRY0</u>					
Status Affected:	<u>C,SZ,Z</u>					
Encoding:	<u>1101</u>	<u>1110</u>	<u>1www</u>	<u>wddd</u>	<u>d000</u>	<u>ssss</u>
Description:	<p><u>Arithmetic shift right the contents of the source register Wb by Wns bits (up to 31 positions), placing the result in the destination register Wnd. Bits that are shifted beyond the rightmost position of the source are stored in the CARRY1 and CARRY0 registers.</u></p> <p><u>The Z and SZ bits will be set if the value placed in Wnd is zero and cleared otherwise. The C bit will be set if any of the bits shifted out were set (in other words, if the resultant CARRY is non-zero) and cleared otherwise.</u></p> <p><u>Note: This instruction operates in word mode only.</u></p>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					

TABLE 20 -- BC: Branch if Carry

Syntax:	<u>{label:}</u>	<u>BRA</u>	<u>C,</u>	<u>Slit16</u>		
	<u>{label:}</u>	<u>BRA</u>	<u>GEU,</u>			
Operands:	<u>Slit16 \in [-32768 ... +32767]</u>					
Operation:	<u>Condition = C</u> <u>If (condition), then (PC+2) + 2*Slit16 -t PC, and NOP + Instruction Register.</u>					
Status Affected:	<u>None</u>					
Encoding:	<u>0011</u>	<u>0001</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>
Description:	<u>If the Carry bit is '1', then the program will branch.</u> <u>The 2's complement number '2*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be (PC+2) + 2*Slit16 . This instruction is then a two-cycle instruction, with a NOP in the second cycle.</u> <u>The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).</u>					
Words:	<u>1</u>					
Cycles:	<u>1 (2)</u>					
Example:		<u>BRA</u>	<u>C, label</u>	<u>: Branch if Carry</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 21 -- BCLRF: Bit Clear f

<u>Syntax:</u>	<u>{label:}</u>	<u>BCLR.b</u>	<u>f,</u>	<u>bit3</u>		
<u>Operands:</u>	<u>bit3 ∈ [0 ... 7]; f ∈ [0 ... 8191]</u>					
<u>Operation:</u>	<u>0 → f<bit3></u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>1010</u>	<u>1001</u>	<u>bbbf</u>	<u>ffff</u>	<u>ffff</u>	<u>ffff</u>
<u>Description:</u>	<u>Bit 'bit3' in file register f is cleared.</u>					
	<u>The 'b' bits select value bit3 of the bit position to be cleared.</u>					
	<u>The 'f' bits select the address of the file register.</u>					
	<u>Note: This instruction operates in byte mode only.</u>					
	<u>Note: The .b extension must be included with the opcode.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1</u>					
<u>Example:</u>		<u>BCLR.b</u>	<u>RAM135, #5</u>	<u>; Clear bit 5 in RAM135</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 22 -- BGE: Branch if Signed Greater Than or Equal

<u>Syntax:</u>	<u>{label;}</u>	<u>BRA</u>	<u>GE,</u>	<u>Slit16</u>		
<u>Operands:</u>	<u>Slit16 \in [-32768 ... +32767]</u>					
<u>Operation:</u>	<u>Condition = (N&&OV) (!N&&!OV)</u> <u>If (Condition), then (PC+2) + 2*Slit16 \rightarrow PC, and NOP \rightarrow Instruction Register.</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>0011</u>	<u>1101</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>
<u>Description:</u>	<u>If the branch condition is met, then the program will branch.</u> <u>The 2's complement number '2*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be (PC+2) + 2*Slit16 . This instruction is then a two-cycle instruction, with a NOP in the second cycle.</u> <u>The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1 (2)</u>					
<u>Example:</u>		<u>BRA</u>	<u>GE, label</u>	<u>; Branch if Greater Than or Equal</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 23 -- BGT: Branch if Signed Greater Than

<u>Syntax:</u>	<u>{label:}</u>	<u>BRA</u>	<u>GT,</u>	<u>Slit16</u>		
<u>Operands:</u>	<u>Slit16 ∈ [-32768 ... +32767]</u>					
<u>Operation:</u>	<u>Condition = (!Z&&N&&OV) (!Z&&!N&&!OV);</u> <u>If (Condition), then (PC+2) + 2*Slit16 → PC, and NOP → Instruction Register.</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>0011</u>	<u>1100</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>
<u>Description:</u>	<u>If the branch condition is met, then the program will branch.</u> <u>The 2's complement number '2*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be (PC+2) + 2*Slit16 . This instruction is then a two-cycle instruction, with a NOP in the second cycle.</u> <u>The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1 (2)</u>					
<u>Example:</u>		<u>BRA</u>	<u>GT, label</u>	<u>: Branch if Greater Than</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 24 -- BGTU: Branch if Unsigned Greater Than

<u>Syntax:</u>	<u>{label:}</u>	<u>BRA</u>	<u>GTU,</u>	<u>Slit16</u>		
<u>Operands:</u>	<u>Slit16 ∈ [-32768 ... +32767]</u>					
<u>Operation:</u>	<u>Condition = (C&&I_Z);</u> <u>If (Condition), then (PC+2) + 2*Slit16 → PC, and NOP → Instruction Register.</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>0011</u>	<u>1110</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>
<u>Description:</u>	<u>If the branch condition is met, then the program will branch.</u> <u>The 2's complement number '2*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be (PC+2) + 2*Slit16. This instruction is then a two-cycle instruction, with a NOP in the second cycle.</u> <u>The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1 (2)</u>					
<u>Example:</u>		<u>BRA</u>	<u>GTU, label</u>	<u>: Branch if Unsigned Greater Than</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 25 -- BLE: Branch if Signed Less Than or Equal

<u>Syntax:</u>	<u>{label:}</u>	<u>BRA</u>	<u>LE,</u>	<u>Slit16</u>		
<u>Operands:</u>	<u>Slit16 ∈ [-32768... +32767]</u>					
<u>Operation:</u>	<u>Condition = Z (N&&!OV) (!N&&OV);</u> <u>If (Condition), then (PC+2) + 2*Slit16 → PC, and NOP → Instruction Register.</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>0011</u>	<u>0100</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>
<u>Description:</u>	<u>If the branch condition is met, then the program will branch.</u> <u>The 2's complement number '2*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be (PC+2) + 2*Slit16 . This instruction is then a two-cycle instruction, with a NOP in the second cycle.</u> <u>The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1 (2)</u>					
<u>Example:</u>		<u>BRA</u>	<u>LE, label</u>	<u>: Branch if Less Than or Equal</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 26 -- BLEU: Branch if Unsigned Less Than or Equal

<u>Syntax:</u>	<u>{label:}</u>	<u>BRA</u>	<u>LEU,</u>	<u>Slit16</u>		
<u>Operands:</u>	<u>Slit16 ∈ [-32768 ... +32767]</u>					
<u>Operation:</u>	<u>Condition = !C Z;</u> <u>If (Condition), then (PC+2) + 2*Slit16 → PC, and NOP → Instruction Register.</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>0011</u>	<u>0110</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>
<u>Description:</u>	<u>If the branch condition is met, then the program will branch.</u> <u>The 2's complement number '2*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be (PC+2) + 2*Slit16 . This instruction is then a two-cycle instruction, with a NOP in the second cycle.</u> <u>The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1 (2)</u>					
<u>Example:</u>		<u>BRA</u>	<u>LEU, label</u>	<u>; Branch if Unsigned Less Than or Equal</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 27 -- BLT: Branch if Signed Less Than

Syntax:	<u>{label:}</u>	<u>BRA</u>	<u>LT,</u>	<u>Slit16</u>		
Operands:	<u>Slit16 ∈ [-32768 ... +32767]</u>					
Operation:	<u>Condition = (N&&!OV) (!N&&OV);</u> <u>If (Condition), then (PC+2) + 2*Slit16 → PC, and NOP → Instruction Register.</u>					
Status Affected:	<u>None</u>					
Encoding:	<u>0011</u>	<u>0101</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>
Description:	<u>If the branch condition is met, then the program will branch.</u> <u>The 2's complement number '2*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be (PC+2) + 2*Slit16. This instruction is then a two-cycle instruction, with a NOP in the second cycle.</u> <u>The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).</u>					
Words:	<u>1</u>					
Cycles:	<u>1 (2)</u>					
Example:		<u>BRA</u>	<u>LT, label</u>	<u>: Branch if Less Than</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 28 -- BN: Branch if Negative

Syntax:	<u>{label:}</u>	<u>BRA</u>	<u>N,</u>	<u>Slit16</u>		
Operands:	<u>Slit16 \in [-32768 ... +32767]</u>					
Operation:	<u>Condition = N</u> <u>If (condition), then (PC+2) + 2*Slit16 \rightarrow PC, and NOP \rightarrow Instruction Register.</u>					
Status Affected:	<u>None</u>					
Encoding:	<u>0011</u>	<u>0011</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>
Description:	<u>If the Negative Flag is '1', then the program will branch.</u> <u>The 2's complement number '2*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be (PC+2) + 2*Slit16. This instruction is then a two-cycle instruction, with a NOP in the second cycle.</u> <u>The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).</u>					
Words:	<u>1</u>					
Cycles:	<u>1 (2)</u>					
Example:		<u>BRA</u>	<u>N, label</u>	<u>: Branch if Negative</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 29 -- BNC: Branch if Not Carry

<u>Syntax:</u>	<u>{label:}</u>	<u>BRA</u>	<u>NC,</u>	<u>Slit16</u>		
	<u>{label:}</u>	<u>BRA</u>	<u>LTU,</u>			
<u>Operands:</u>	<u>Slit16 ∈ [-32768 ... +32767]</u>					
<u>Operation:</u>	<u>Condition = !C</u> <u>If (condition), then (PC+2) + 2*Slit16 → PC, and NOP → Instruction Register.</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>0011</u>	<u>1001</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>
<u>Description:</u>	<u>If the Carry bit is '0', then the program will branch.</u> <u>The 2's complement number '2*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be (PC+2) + 2*Slit16 . This instruction is then a two-cycle instruction, with a NOP in the second cycle.</u> <u>The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1 (2)</u>					
<u>Example:</u>		<u>BRA</u>	<u>NC, label</u>	<u>: Branch if Not Carry</u>		
	<u>Before Instruction</u> <u>After Instruction</u>					

TABLE 30 -- BNN: Branch if Not Negative

<u>Syntax:</u>	<u>{label:}</u>	<u>BRA</u>	<u>NN,</u>	<u>Slit16</u>		
<u>Operands:</u>	<u>Slit16 ∈ [-32768 ... +32767]</u>					
<u>Operation:</u>	<u>Condition = !N</u> <u>If (condition), then (PC+2) + 2*Slit16 → PC, and NOP → Instruction Register.</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>0011</u>	<u>1011</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>
<u>Description:</u>	<u>If the Negative Flag is '0', then the program will branch.</u> <u>The 2's complement number '2*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be (PC+2) + 2*Slit16. This instruction is then a two-cycle instruction, with a NOP in the second cycle.</u> <u>The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1 (2)</u>					
<u>Example:</u>		<u>BRA</u>	<u>NN, label</u>	<u>: Branch if Not Negative</u>		
	<u>Before Instruction</u> <u>After Instruction</u>					

TABLE 31 -- BNOV: Branch if Not Overflow

<u>Syntax:</u>	<u>{label:}</u>	<u>BRA</u>	<u>NOV,</u>	<u>Slit16</u>		
<u>Operands:</u>	<u>Slit16 ∈ [-32768 ... +32767]</u>					
<u>Operation:</u>	<u>Condition = !OV</u> <u>If (condition), then (PC+2) + 2*Slit16 → PC, and NOP → Instruction Register.</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>0011</u>	<u>1000</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>
<u>Description:</u>	<u>If the Overflow Flag is '0', then the program will branch.</u> <u>The 2's complement number '2*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be (PC+2) + 2*Slit16. This instruction is then a two-cycle instruction, with a NOP in the second cycle.</u> <u>The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1 (2)</u>					
<u>Example:</u>		<u>BRA</u>	<u>NOV, label</u>	<u>; Branch if Not OVerflow</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 32 -- BNZ: Branch if Not Zero

<u>Syntax:</u>	<u>{label:}</u>	<u>BRA</u>	<u>NZ,</u>	<u>Slit16</u>		
<u>Operands:</u>	<u>Slit16 ∈ [-32768 ... +32767]</u>					
<u>Operation:</u>	<u>Condition = !Z</u> <u>If (condition), then (PC+2) + 2*Slit16 → PC, and NOP → Instruction Register.</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>0011</u>	<u>1010</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>
<u>Description:</u>	<u>If the Zero Flag is '0', then the program will branch.</u> <u>The 2's complement number '2*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be (PC+2) + 2*Slit16. This instruction is then a two-cycle instruction, with a NOP in the second cycle.</u> <u>The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1 (2)</u>					
<u>Example:</u>		<u>BRA</u>	<u>NZ, label</u>	<u>: Branch if Not Zero</u>		
	<u>Before Instruction</u> <u>After Instruction</u>					

TABLE 33 -- BOA: Branch if Overflow Accumulator A

<u>Syntax:</u>	<u>{label:}</u>	<u>BRA</u>	<u>OA,</u>	<u>Slit16</u>		
<u>Operands:</u>	<u>Slit16 ∈ [-32768... +32767]</u>					
<u>Operation:</u>	<u>Condition = OA</u> <u>If (condition), then (PC+2) + 2*Slit16 → PC, and NOP → Instruction Register.</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>0000</u>	<u>1100</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>
<u>Description:</u>	<u>If the OA Flag is '1', then the program will branch.</u> <u>The 2's complement number '2*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be (PC+2) + 2*Slit16 . This instruction is then a two-cycle instruction, with a NOP in the second cycle.</u> <u>The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1 (2)</u>					
<u>Example:</u>		<u>BRA</u>	<u>OA, label</u>	<u>: Branch if Accumulator A Overflow</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 34 -- BOB: Branch if Overflow Accumulator B

Syntax:	<u>{label:}</u>	<u>BRA</u>	<u>OB,</u>	<u>Slit16</u>		
Operands:	<u>Slit16 \in [-32768 ... +32767]</u>					
Operation:	<u>Condition = OB</u> <u>If (condition), then (PC+2) + 2*Slit16 \rightarrow PC, and NOP \rightarrow Instruction Register.</u>					
Status Affected:	<u>None</u>					
Encoding:	<u>0000</u>	<u>1101</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>
Description:	<u>If the OB Flag is '1', then the program will branch.</u> <u>The 2's complement number '2*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be (PC+2) + 2*Slit16. This instruction is then a two-cycle instruction, with a NOP in the second cycle.</u> <u>The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).</u>					
Words:	<u>1</u>					
Cycles:	<u>1 (2)</u>					
Example:		<u>BRA</u>	<u>OB, label</u>	<u>; Branch if Accumulator B Overflow</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 35 -- BOV: Branch if Overflow

<u>Syntax:</u>	<u>{label:}</u>	<u>BRA</u>	<u>OV,</u>	<u>Slit16</u>		
<u>Operands:</u>	<u>Slit16 ∈ [-32768 ... +32767]</u>					
<u>Operation:</u>	<u>Condition = OV</u> <u>If (condition), then (PC+2) + 2*Slit16 → PC, and NOP → Instruction Register.</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>0011</u>	<u>0000</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>
<u>Description:</u>	<u>If the Overflow Flag is '1', then the program will branch.</u> <u>The 2's complement number '2*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be (PC+2) + 2*Slit16 . This instruction is then a two-cycle instruction, with a NOP in the second cycle.</u> <u>The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1 (2)</u>					
<u>Example:</u>		<u>BRA</u>	<u>OV, label</u>	<u>: Branch if Overflow</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 36 -- BRA: Branch Unconditionally

<u>Syntax:</u>	<u>{label:}</u>	<u>BRA</u>	<u>Slit16</u>			
<u>Operands:</u>	<u>Slit16 ∈ [-32768 ... +32767]</u>					
<u>Operation:</u>	<u>(PC+2) + 2*Slit16 → PC, and NOP → Instruction Register.</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>0011</u>	<u>0111</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>
<u>Description:</u>	<u>The program will branch unconditionally.</u> <u>The 2's complement number '2*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be (PC+2) + 2*Slit16. This instruction is then a two-cycle instruction, with a NOP in the second cycle.</u> <u>The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>2</u>					
<u>Example:</u>		<u>BRA</u>	<u>label</u>	<u>; Branch unconditionally</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 37 -- BRAW: Computed Branch

<u>Syntax:</u>	<u>{label:}</u>	<u>BRA</u>	<u>Wn</u>			
<u>Operands:</u>	<u>Wn ∈ [W0 ... W15]</u>					
<u>Operation:</u>	<u>(PC) +2 + (2 * (Wn)) → PC, NOP → Instruction Register.</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>0000</u>	<u>0001</u>	<u>0110</u>	<u>0000</u>	<u>0000</u>	<u>ssss</u>
<u>Description:</u>	<u>Computed branch with a jump up to 32K instructions forward or backward from the current location.</u>					
	<u>The sign extended 17-bit value (2 * (Wn)) is added to the contents of the PC and the result is stored into the PC. BRAW is a two-cycle instruction.</u>					
	<u>The 's' bits select the address of the source register.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>2</u>					
<u>Example:</u>		<u>BRA</u>	<u>W11</u>	<u>; Branch to PC+W11</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 38 -- BSA: Branch if ACCA Saturation

<u>Syntax:</u>	<u>{label:}</u>	<u>BRA</u>	<u>SA,</u>	<u>Slit16</u>		
<u>Operands:</u>	<u>Slit16 ∈ [-32768 ... +32767]</u>					
<u>Operation:</u>	<u>Condition = SA</u> <u>If (condition), then (PC+2) + 2*Slit16 → PC, and NOP → Instruction Register.</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>0000</u>	<u>1110</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>
<u>Description:</u>	<u>If the ACCA Saturation Flag is '1', then the program will branch.</u> <u>The 2's complement number '2*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be (PC+2) + 2*Slit16. This instruction is then a two-cycle instruction, with a NOP in the second cycle.</u> <u>The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1 (2)</u>					
<u>Example:</u>		<u>BRA</u>	<u>SA, label</u>	<u>: Branch if ACCA Saturation</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 39 -- BSB: Branch if ACCB Saturation

<u>Syntax:</u>	<u>{label:}</u>	<u>BRA</u>	<u>SB,</u>	<u>Slit16</u>		
<u>Operands:</u>	<u>Slit16 \in [-32768 ... +32767]</u>					
<u>Operation:</u>	<u>Condition = SB</u> <u>if (condition), then (PC+2) + 2*Slit16\rightarrow PC, and NOP \rightarrow Instruction Register.</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>0000</u>	<u>1111</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>
<u>Description:</u>	<u>If the ACCB Saturation Flag is '1', then the program will branch.</u> <u>The 2's complement number '2*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be (PC+2) + n. This instruction is then a two-cycle instruction, with a NOP in the second cycle.</u> <u>The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1 (2)</u>					
<u>Example:</u>		<u>BRA</u>	<u>SB, label</u>	<u>; Branch if ACCB Saturation</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 40 -- BSETF: Bit Set f

Syntax:	{label:}	BSET.b	f	bit3		
Operands:	bit3 ∈ [0 ... 7]; f ∈ [0 ... 8191]					
Operation:	1 → f<bit3>					
Status Affected:	None					
Encoding:	1010	1000	bbbf	ffff	ffff	ffff
Description:	Bit 'bit3' in file register f is set. The 'b' bits select value bit3 of the bit position to be cleared. The 'f' bits select the address of the file register. Note: This instruction operates in byte mode only. Note: The .b extension must be included with the opcode.					
Words:	1					
Cycles:	1					
Example:		BSET.B	RAM135, #5	; Set bit 5 in RAM135		
	Before Instruction					
	After Instruction					

TABLE 41 -- BSW: Bit Write in Ws

Syntax:	{label:}	<u>BSW.C</u>	<u>Ws</u>	<u>Wb</u>		
		<u>BSW.Z</u>	<u>[Ws]</u>			
			<u>[Ws]++</u>			
			<u>[Ws]--</u>			
			<u>[Ws++]</u>			
			<u>[Ws--]</u>			
Operands:	<u>Wb ∈ [W0 ... W15]; Ws ∈ [W0 ... W15]</u>					
Operation:	<u>If “.Z” option, then $\overline{Z} \rightarrow \Omega\sigma<(\Omega\beta)>$</u> <u>$\square\square\square\square\square\square\square\square\square\square\square\square\square\square\square\square \rightarrow Ws<(Wb)>$</u>					
Status Affected:	<u>None</u>					
Encoding:	<u>1010</u>	<u>1101</u>	<u>Zwww</u>	<u>w000</u>	<u>0ppp</u>	<u>ssss</u>
Description:	<u>Bit (Wb) in register Ws is written with the value of the C or Z bit.</u> <u>The ‘w’ bits select the address of the bit select register.</u> <u>The ‘Z’ bit selects the Z or C flag bit as source.</u> <u>The ‘s’ bits select the address of the source register.</u> <u>The ‘p’ bits select source address mode 2.</u> <u>See Table 1-5 for modifier addressing information.</u>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>BSW.Z</u>	<u>W5,W6</u>	<u>; Test/Set bit</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 42 -- BTFSC: Bit Test f, Skip if Clear

<u>Syntax:</u>	<u>{label:}</u>	<u>BTSC.b</u>	<u>f</u>	<u>bit3</u>		
<u>Operands:</u>	<u>bit3 ∈ [0 ... 7]; f ∈ [0 ... 8191]</u>					
<u>Operation:</u>	<u>Test (f)<bit3>, skip if clear</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>1010</u>	<u>1111</u>	<u>bbbf</u>	<u>ffff</u>	<u>ffff</u>	<u>ffff</u>
<u>Description:</u>	<u>Bit 'bit3' in (f) is tested. If the bit is '0', then the fetched instruction is discarded and on the next cycle a NOP is executed instead.</u> <u>The 'b' bits select the value bit3 of the bit position to be tested.</u> <u>The 'f' bits select the address of the file register.</u> <u>Note: This instruction operates in byte mode only.</u> <u>Note: The .b extension must be included with the opcode.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1 (2 or 3)</u>					
<u>Example:</u>		<u>BTSC.b</u>	<u>RAM135, #5</u>	<u>; Bit test bit 5 in RAM135, skip if clear</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 43 -- BTFSS: Bit Test f, Skip if Set

<u>Syntax:</u>	<u>{label:}</u>	<u>BTSS.b</u>	<u>f,</u>	<u>bit3</u>		
<u>Operands:</u>	<u>bit3 ∈ [0 ... 7]; f ∈ [0 ... 8191]</u>					
<u>Operation:</u>	<u>Test (f)<bit3>, skip if set</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>1010</u>	<u>1110</u>	<u>bbbf</u>	<u>ffff</u>	<u>ffff</u>	<u>ffff</u>
<u>Description:</u>	<u>Bit 'bit3' in (f) is tested. If the bit is '1', then the fetched instruction is discarded and on the next cycle a NOP is executed instead.</u> <u>The 'b' bits select value bit3 of the bit position to be cleared.</u> <u>The 'f' bits select the address of the file register.</u> <u>Note: This instruction operates in byte mode only.</u> <u>Note: The .b extension must be included with the opcode.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1 (2 or 3)</u>					
<u>Example:</u>		<u>BTSS.b</u>	<u>RAM135, #5</u>	<u>; Bit test bit 5 in RAM135, skip if set</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 44 -- BTGF: Bit Toggle f

<u>Syntax:</u>	<u>{label:}</u>	<u>BTG.b</u>	<u>f,</u>	<u>bit3</u>		
<u>Operands:</u>	<u>bit3 ∈ [0 ... 7]; f ∈ [0 ... 8191]</u>					
<u>Operation:</u>	<u>(f) < bit3 > → (f)<bit3></u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>1010</u>	<u>1010</u>	<u>bbbf</u>	<u>ffff</u>	<u>ffff</u>	<u>ffff</u>
<u>Description:</u>	<u>Bit 'bit3' in file register f is toggled.</u> <u>The 'b' bits select value bit3 of the bit position to be cleared.</u> <u>The 'f' bits select the address of the file register.</u> <u>Note: This instruction operates in byte mode only.</u> <u>Note: The .b extension must be included with the opcode.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1</u>					
<u>Example:</u>		<u>BTG.b</u>	<u>RAM135, #5</u>	<u>; Toggle bit 5 in RAM135</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 45 -- BTSC: Bit Test Ws, Skip if Clear

<u>Syntax:</u>	<u>{label:}</u>	<u>BTSC</u>	<u>Ws,</u>	<u>bit4</u>		
			<u>[Ws],</u>			
			<u>[Ws]++,</u>			
			<u>[Ws]--,</u>			
			<u>[Ws++]</u> ,			
			<u>[Ws--]</u> ,			
<u>Operands:</u>	<u>bit4 ∈ [0 ... 15]; Ws ∈ [W0 ... W15]</u>					
<u>Operation:</u>	<u>Test (Ws)<bit4>, skip if clear.</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>1010</u>	<u>0111</u>	<u>bbbb</u>	<u>0000</u>	<u>0ppp</u>	<u>ssss</u>
<u>Description:</u>	<u>Bit 'bit4' in (Ws) is tested. If the bit is '0', then the fetched instruction is discarded and on the next cycle a NOP is executed instead.</u> <u>The 'b' bits select value bit4 of the bit position to be tested.</u> <u>The 's' bits select the address of the source register.</u> <u>The 'p' bits select source address mode 2 (values 0-4).</u> <u>See Table 1-5 for modifier addressing information.</u> <u>Note: This instruction operates in word mode only.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1 (2 or 3)</u>					
<u>Example:</u>		<u>BTSC</u>	<u>W6, #5</u>	<u>; Test bit 5 in W6, skip if clear</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 46 -- BTSS: Bit Test Ws, Skip if Set

Syntax:	{label:}	BTSS	Ws,	bit4		
			<u>[Ws],</u>			
			<u>[Ws]++.</u>			
			<u>[Ws]--.</u>			
			<u>[Ws++].</u>			
			<u>[Ws--].</u>			
Operands:	<u>bit4 ∈ [0 ... 15]; Ws ∈ [W0 ... W15]</u>					
Operation:	<u>Test (Ws)<bit4>, skip if set.</u>					
Status Affected:	<u>None</u>					
Encoding:	<u>1010</u>	<u>0110</u>	<u>bbbb</u>	<u>0000</u>	<u>0ppp</u>	<u>ssss</u>
Description:	<p><u>Bit ‘bit4’ in (Ws) is tested. If the bit is ‘1’, then the fetched instruction is discarded and on the next cycle a NOP is executed instead.</u></p> <p><u>The ‘b’ bits select the value bit4 of the bit position to be tested.</u></p> <p><u>The ‘s’ bits select the address of the source register.</u></p> <p><u>The ‘p’ bits select source address mode 2 (values 0-4).</u></p> <p><u>See Table 1-5 for modifier addressing information.</u></p> <p><u>Note: This instruction operates in word mode only.</u></p>					
Words:	<u>1</u>					
Cycles:	<u>1 (2 or 3)</u>					
Example:		BTSS	W6, #5	<u>; Test bit 5 in W6, skip if set</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 47 -- BTST: Bit Test in Ws

Syntax:	{label:}	BTST.C	Ws,	bit4		
		BTST.Z	[Ws],			
			[Ws]++,			
			[Ws]--,			
			[Ws++] ,			
			[Ws--] ,			
Operands:	bit4 \in [0 ... 15]; Ws \in [W0 ... W15];					
Operation:	if “ Z ” option, (Ws) < bit4 > \rightarrow Z if “ C ” option, (Ws) < bit4 > \rightarrow C					
Status Affected:	C or Z					
Encoding:	1010	0011	bbbb	z000	0ppp	ssss
Description:	<p>Bit ‘bit4’ in register Ws is tested.</p> <p>The Zero flag contains the inversion of the bit or the Carry flag contains the bit.</p> <p>The ‘b’ bits select value bit4 of the bit position to be test/set.</p> <p>The ‘Z’ bit selects the Z or C flag bit as destination.</p> <p>The ‘s’ bits select the address of the source register.</p> <p>The ‘p’ bits select source address mode 2.</p> <p>See Table 1-5 for modifier addressing information.</p> <p>Note: This instruction operates in word mode only.</p>					
Words:	1					
Cycles:	1					
Example:		BTST.C	W6,#5	; Test bit 5 in W6 to the C flag		
	Before Instruction					
	After Instruction					

TABLE 48 -- BTSTf: Bit Test f

<u>Syntax:</u>	<u>{label:}</u>	<u>BTST.b</u>	<u>f,</u>	<u>bit3</u>		
<u>Operands:</u>	<u>bit3 ∈ [0 ... 7]; f ∈ [0 ... 8191]</u>					
<u>Operation:</u>	<u>(f) < bit3 >→ Z</u>					
<u>Status Affected:</u>	<u>Z</u>					
<u>Encoding:</u>	<u>1010</u>	<u>1011</u>	<u>bbbf</u>	<u>ffff</u>	<u>ffff</u>	<u>ffff</u>
<u>Description:</u>	<u>Bit 'bit3' in file register f is tested, the Zero Flag bit is set if it is zero and cleared otherwise. The file register contents are unchanged.</u> <u>The 'b' bits select value bit3 of the bit position to be cleared.</u> <u>The 'f' bits select the address of the file register.</u> <u>Note: This instruction operates in byte mode only.</u> <u>Note: The .b extension must be included with the opcode.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1</u>					
<u>Example:</u>		<u>BTST.b</u>	<u>RAM135, #5</u>	<u>: Test bit 5 in RAM135</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 49 -- BTSTS: Bit Test/Set in Ws

Syntax:	<u>{label:}</u>	<u>BTSTS.C</u>	<u>Ws,</u>	<u>bit4</u>		
		<u>BTSTS.Z</u>	<u>[Ws],</u>			
			<u>[Ws]++,</u>			
			<u>[Ws]--,</u>			
			<u>[Ws++]</u> ,			
			<u>[Ws--]</u> ,			
Operands:	<u>bit4 ∈ [0 ... 15]; Ws ∈ [W0 ... W15]</u>					
Operation:	<u>if “.Z” option, first (Ws)<bit4> → Z, then 1 → Ws<bit4></u> <u>if “.C” option, first (Ws)<bit4> → C, then 1 → Ws<bit4></u>					
Status Affected:	<u>C or Z</u>					
Encoding:	<u>1010</u>	<u>0100</u>	<u>bbbb</u>	<u>z000</u>	<u>0ppp</u>	<u>ssss</u>
Description:	<u>Bit ‘bit4’ in register Ws is tested and then set.</u> <u>The ‘b’ bits select the value bit4 of the bit position to be test/set.</u> <u>The ‘Z’ bit selects the Z or C flag bit as destination.</u> <u>The ‘s’ bits select the address of the source register.</u> <u>The ‘p’ bits select source address mode 2.</u> <u>See Table 1-5 for modifier addressing information.</u> <u>Note: This instruction operates in word mode only.</u>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>BTSTS.Z</u>	<u>W6,#5</u>	<u>: Test/Set bit 5 in W6 to the Z flag</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 50 -- BTSTSF: Bit Test/Set f

<u>Syntax:</u>	<u>{label:}</u>	<u>BTSTS.b</u>	<u>f,</u>	<u>bit3</u>		
<u>Operands:</u>	<u>bit3 ∈ [0 ... 7]; f ∈ [0 ... 8191]</u>					
<u>Operation:</u>	<u>First(f) < bit3 > → Z, then 1 → (f)<bit3></u>					
<u>Status Affected:</u>	<u>Z</u>					
<u>Encoding:</u>	<u>1010</u>	<u>1100</u>	<u>bbbf</u>	<u>ffff</u>	<u>ffff</u>	<u>ffff</u>
<u>Description:</u>	<u>Bit 'bit3' in file register f is tested and then set.</u> <u>The 'b' bits select value bit3 of the bit position to be cleared.</u> <u>The 'f' bits select the address of the file register.</u> <u>Note: This instruction operates in byte mode only.</u> <u>Note: The .b extension must be included with the opcode.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1</u>					
<u>Example:</u>		<u>BTSTS.b</u>	<u>RAM135, #5</u>	<u>; Test/Set bit 5 in RAM135</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 51 -- BTSTW: Bit Test in Ws

Syntax:	{label:}	BTST.C	Ws,	Wb		
		BTST.Z	[Ws],			
			[Ws]++,			
			[Ws]--,			
			[Ws++]			
			[Ws--],			
Operands:	Wb ∈ [W0 ... W15]; Ws ∈ [W0 ... W15]					
Operation:	if “.Z” option, (Ws)<(Wb)> → Z if “.C” option, (Ws)<(Wb)> → C					
Status Affected:	C or Z					
Encoding:	1010	0101	Zwww	w000	0ppp	ssss
Description:	<u>Bit (Wb) in register Ws is tested.</u> <u>The Zero flag contains the inversion of the bit or the Carry flag contains the bit.</u> <u>The ‘w’ bits select the address of the bit select register.</u> <u>The ‘Z’ bit selects the Z or C flag bit as destination.</u> <u>The ‘s’ bits select the address of the source register.</u> <u>The ‘p’ bits select source address mode 2.</u> <u>See Table 1-5 for modifier addressing information.</u>					
Words:	1					
Cycles:	1					
Example:		BTST.C	W5,W6	: Test bit in W5 selected by W6		
	Before Instruction					
	After Instruction					

TABLE 52 -- BZ: Branch if Zero

<u>Syntax:</u>	<u>{label:}</u>	<u>BRA</u>	<u>BZ,</u>	<u>Slit16</u>		
<u>Operands:</u>	<u>Slit16 \in [-32768 ... +32767]</u>					
<u>Operation:</u>	<u>Condition = Z</u> <u>if (condition), then (PC+2) + 2*Slit16 \rightarrow PC, and NOP \rightarrow Instruction Register.</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>0011</u>	<u>0010</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>
<u>Description:</u>	<u>If the Z Flag is '1', then the program will branch.</u> <u>The 2's complement number '2*Slit16' (the offset) is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be (PC+2) + 2*Slit16. This instruction is then a two-cycle instruction, with a NOP in the second cycle.</u> <u>The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+2).</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1 (2)</u>					
<u>Example:</u>		<u>BRA</u>	<u>Z, label</u>	<u>: Branch if Zero</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 53 -- CALL: Call Subroutine

Syntax:	<u>{label:}</u>	<u>CALL</u>	<u>lit23</u>			
		<u>CALLS</u>				
Operands:	<u>lit23</u> \in [0 ... 8388606]					
Operation:	<u>(PC) +4 \rightarrow PC,</u> <u>(PC<15:0>) \rightarrow TOS,</u> <u>(W15)+2 \rightarrow W15</u> <u>(PC<23:16>) \rightarrow TOS,</u> <u>(W15)+2 \rightarrow W15</u> <u>lit23 \rightarrow PC, NOP \rightarrow Instruction Register.</u> <u>If S = 1, copy the contents of the primary registers into the shadow registers.</u>					
Status Affected:	<u>None</u>					
Encoding:						
1st word	<u>0000</u>	<u>001S</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnn0</u>
2nd word	<u>0000</u>	<u>0000</u>	<u>0000</u>	<u>0000</u>	<u>0nnn</u>	<u>nnnn</u>
Description:	<u>Subroutine call of entire 4M instruction program memory range. First, return</u> <u>address (PC+4) is pushed onto the return stack (24-bits wide).</u> <u>Then the 24-bit value 'lit23' is loaded into the PC. CALL is a two-cycle</u> <u>instruction.</u> <u>The 'n' bits form the target address.</u> <u>If 'S' = 1, the primary registers are copied into the shadow registers.</u> <u>If 'S' = 0, no update occurs.</u>					
Words:	<u>2</u>					
Cycles:	<u>2</u>					
Example:		<u>CALL</u>	<u>label</u>	<u>; Call subroutine</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 54 -- CALLW: Call Indirect Subroutine

<u>Syntax:</u>	<u>{label:}</u>	<u>CALL</u>	<u>Wn</u>			
		<u>CALL.S</u>				
<u>Operands:</u>	<u>Wn ∈ [W0, W15]</u>					
<u>Operation:</u>	<u>(PC) +2 → PC,</u> <u>(PC<15:0>) → TOS,</u> <u>(W15)+2 → W15</u> <u>(PC<23:16>) → TOS,</u> <u>(W15)+2 → W15</u> <u>0 → PC<22:17>, (Wn) → PC<16:1>, 0 → PC<0>;</u> <u>NOP → Instruction Register.</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>0000</u>	<u>0001</u>	<u>S000</u>	<u>0000</u>	<u>0000</u>	<u>ssss</u>
<u>Description:</u>	<u>Indirect subroutine call of first 64K instructions of program memory. First,</u> <u>return address (PC+2) is pushed onto the return stack.</u> <u>Then, the 16-bit value (Wn) is left shifted 1 bit, zero-extended and loaded into</u> <u>the PC. CALL is a two-cycle instruction.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>2</u>					
<u>Example:</u>		<u>CALL</u>	<u>W5</u>	<u>; Call indirect subroutine</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 55 -- CLRAC: Clear Accumulator, Prefetch Operands

<u>Syntax:</u>	<u>{label:}</u>	<u>CLR</u>	<u>A,</u>	<u>Wxp,[Wx]</u>	<u>Wyp,[Wy]</u>	<u>AWB</u>
			<u>B,</u>	<u>Wxp,[Wx]+=kx</u>	<u>Wyp,[Wy]+=ky</u>	<u>none</u>
				<u>Wxp,[Wx]-=kx ‡</u>	<u>Wyp,[Wy]-=ky ‡</u>	
				<u>Wxp,[W5+W8]</u>	<u>Wyp,[W7+W8]</u>	
				<u>none</u>	<u>none</u>	
				<u>‡ Alternate format for negative kx,ky</u>		
<u>Operands:</u>	<u>Wxp ∈ {W0 ... W3}; Wx ∈ {W4, W5}; kx ∈ {-6, -4, -2, 2, 4, 6};</u> <u>Wyp ∈ {W0 ... W3}; Wy ∈ {W6, W7}; ky ∈ {-6, -4, -2, 2, 4, 6};</u> <u>AWB ∈ {W9, [W9]++}</u>					
<u>Operation:</u>	<u>0 → ACC(A or B)</u> <u>([Wx]) → Wxp; (Wx)+kx → Wx;</u> <u>([Wy]) → Wyp; (Wy)+ky → Wy;</u> <u>(ACC(B or A)) rounded → AWB</u>					
<u>Status Affected:</u>	<u>OA, OB, SA, SB</u>					
<u>Encoding:</u>	<u>1100</u>	<u>0011</u>	<u>A0xx</u>	<u>yyii</u>	<u>iijj</u>	<u>jjaa</u>
<u>Description:</u>	<u>Clear the specified accumulator, prefetch operands and optionally store accumulator results in preparation for a repeated MAC type instruction.</u> <u>Wx register specifies the prefetch of the multiplier Wxp register. The prefetch is done with indirect, indirect with post inc/dec, indirect with register offset, copy of the other prefetch or none. Post-modify Wx as required.</u> <u>Wy register specifies the prefetch of the multiplier Wyp register. Post-modify Wy as required.</u> <u>AWB specifies the direct or indirect store of the convergently rounded contents of other accumulator, if required. Note that the specification of (B or A) is consistent with the MAC instruction. For example, CLRAC A, W9 will store ACCB into W9.</u> <u>The 'A' bit selects the other accumulator used for write back.</u> <u>The 'i' bits select the Wx pre-fetch operation.</u> <u>The 'j' bits select the Wy pre-fetch operation.</u> <u>The 'x' bits select the pre-fetch Wxp destination.</u> <u>The 'y' bits select the pre-fetch Wyp destination.</u> <u>The 'a' bits select the accumulator write-back destination.</u> <u>See Table 1-9 through Table 1-14 for modifier addressing information.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1</u>					
<u>Example:</u>	<u>CLR</u>	<u>A,W0,[W4]-</u>			<u>: Clear ACCA, prefetch, move</u>	
		<u>=6,W1,[W6],[W9]++</u>			<u>ACCB to [W9]++</u>	
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 56 -- CLRF: Clear f or Ww

<u>Syntax:</u>	<u>{label:}</u>	<u>CLR{.b}</u>	<u>f</u>			
			<u>Ww</u>			
<u>Operands:</u>	<u>f ∈ [0 ... 8191]</u>					
<u>Operation:</u>	<u>0 → destination designated by D</u>					
<u>Status Affected:</u>	<u>Z</u>					
<u>Encoding:</u>	<u>1110</u>	<u>1111</u>	<u>0Bdf</u>	<u>ffff</u>	<u>ffff</u>	<u>ffff</u>
<u>Description:</u>	<u>Clear the register designated by D: If the optional Ww is specified, D=0 and clear Ww; otherwise, D=1 and clear the file register. Z flag is set.</u> <u>The 'B' bit selects byte or word operation.</u> <u>The 'f' bits select the address of the file register.</u> <u>The 'D' bit selects the destination.</u> <u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1</u>					
<u>Example:</u>		<u>CLR</u>	<u>345</u>	<u>; Clear file register 345</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 57 -- CLRWDT: Clear Watchdog Timer

<u>Syntax:</u>	<u>{label:}</u>	<u>CLRWDT</u>				
<u>Operands:</u>	<u>none</u>					
<u>Operation:</u>	<u>0 → WDT Reg</u>					
<u>Status Affected:</u>	<u>IQ PD</u>					
<u>Encoding:</u>	<u>1111</u>	<u>1110</u>	<u>0110</u>	<u>0000</u>	<u>0000</u>	<u>0000</u>
<u>Description:</u>	<u>Clear the WatchDog Timer register.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1</u>					
<u>Example:</u>		<u>CLRWDT</u>		<u>: Clear Watchdog Timer</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 58 -- COM: Complement Ws

Syntax:	{label:}	COM{.b}	Ws,	Wd		
			[Ws],	[Wd]		
			[Ws]++,	[Wd]++		
			[Ws]--,	[Wd]--		
			[Ws++]	[Wd++]		
			[Ws--]	[Wd--]		
Operands:	Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]					
Operation:	(Ws) → Wd					
Status Affected:	Z, N					
Encoding:	1110	1010	1Bqq	qddd	dppp	ssss
Description:	<p>Compute the 1's complement of the contents of the source register Ws and place the result in the destination register Wd.</p> <p>The 'B' bit selects byte or word operation. The 's' bits select the address of the source register. The 'd' bits select the address of the destination register. The 'p' bits select the source address mode 2 (values 0-4). The 'q' bits select the destination address mode 2 (values 0-4).</p> <p>See Table 1-5 and Table 1-6 for modifier addressing information.</p> <p>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</p>					
Words:	1					
Cycles:	1					
Example:		COM	W5,W7	; Complement		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 59 -- COMF: Complement f

Syntax:	{label:}	COM{.b}	f	{,Ww}		
Operands:	f ∈ [0 ... 8191]					
Operation:	(f) → destination designated by D					
Status Affected:	Z, N					
Encoding:	1110	1110	1BDf	ffff	ffff	ffff
Description:	<p>Compute the 1's complement of the contents of the file register and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.</p> <p>The 'B' bit selects byte or word operation. The 'f' bits select the address of the file register. The 'D' bit selects the destination.</p> <p>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</p>					
Words:	1					
Cycles:	1					
Example:		COMF	RAM135	: Complement		
	Before Instruction					
	After Instruction					

TABLE 60 -- CP: Compare Wb with Ws, Set status flags

Syntax:	{label:}	CP{.b}	Wb,	Ws		
				<u>[Ws]</u>		
				<u>[Ws]++</u>		
				<u>[Ws]--</u>		
				<u>[Ws++]</u>		
				<u>[Ws--]</u>		
Operands:	Wb ∈ [W0 ... W15]; Ws ∈ [W0 ... W15]					
Operation:	(Ws) - (Wb)					
Status Affected:	C, DC, N, OV, Z					
Encoding:	<u>1110</u>	<u>0001</u>	<u>0www</u>	<u>wB00</u>	<u>0ppp</u>	<u>ssss</u>
Description:	<p><u>Compute (Ws) - (Wb), equivalent to SUBR instruction, then set flags but do not store result.</u></p> <p><u>The 'B' bit selects byte or word operation.</u></p> <p><u>The 'p' bits select source address mode 2.</u></p> <p><u>The 'w' bits select the address of the Wb source register.</u></p> <p><u>The 's' bits select the address of the Ws source register.</u></p> <p><u>See Table 1-5 for modifier addressing information.</u></p> <p><u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>CP</u>	<u>W5,W6</u>	<u>; Skip</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 61 -- CPB: Compare Wb with Ws with Borrow, set status flags

Syntax:	{label:}	CPB{.b}	Wb,	Ws		
				<u>[Ws]</u>		
				<u>[Ws]++</u>		
				<u>[Ws]--</u>		
				<u>[Ws++]</u>		
				<u>[Ws--]</u>		
Operands:	<u>Wb</u> ∈ [W0 ... W15]; <u>Ws</u> ∈ [W0 ... W15]					
Operation:	<u>(Ws) - (Wb) - (C)</u>					
Status Affected:	<u>C, DC, N, OV, Z</u>					
Encoding:	<u>1110</u>	<u>0001</u>	<u>1www</u>	<u>wB00</u>	<u>0ppp</u>	<u>ssss</u>
Description:	<p><u>Compute (Ws) - (Wb) - (c), equivalent to SUBRB instruction, then set flags but do not store result.</u></p> <p><u>The 'B' bit selects byte or word operation.</u></p> <p><u>The 'p' bits select source address mode 2.</u></p> <p><u>The 'w' bits select the address of the Wb source register.</u></p> <p><u>The 's' bits select the address of the Ws source register.</u></p> <p><u>See Table 1-5 for modifier addressing information.</u></p> <p><u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>CPB</u>	<u>W5,W6</u>	<u>; Skip</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 62 -- CPBLS: Compare Wb with lit5 with borrow, Set status flags

Syntax:	{label:}	CPB{.b}	Wb,	lit5		
Operands:	Wb \in [W0 ... W15]; lit5 \in [0 ... 31]					
Operation:	(Wb) - lit5 - C)					
Status Affected:	C, DC, N, OV, Z					
Encoding:	1110	0001	1www	wB00	011k	kkkk
Description:	<p>Compute (Wb) - lit5, set flags but do not store result.</p> <p>The 'B' bit selects byte or word operation.</p> <p>The 'w' bits select the address of the Wb source register.</p> <p>The 'k' bits provide the literal operand, a five bit integer number.</p> <p>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</p>					
Words:	1					
Cycles:	1					
Example:		CPB	W5, #30			
	Before Instruction					
	After Instruction					

TABLE 63 -- CPF: Compare f with Ww, Set status flags

Syntax:	<u>{label:}</u>	<u>CP{.b}</u>	<u>f</u>			
Operands:	<u>f ∈ [0 ... 8191]</u>					
Operation:	<u>(f) - (Ww)</u>					
Status Affected:	<u>C, DC, N, OV, Z</u>					
Encoding:	<u>1110</u>	<u>0011</u>	<u>0B0f</u>	<u>ffff</u>	<u>ffff</u>	<u>ffff</u>
Description:	<u>Compute (f) - (Wd), set flags but do not store result.</u> <u>The 'B' bit selects byte or word operation.</u> <u>The 'f' bits select the address of the file register.</u> <u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>CP</u>	<u>RAM135</u>	<u>: Compare</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 64 -- CPF0: Compare f with 0x0000, Set status flags

Syntax:	{label:}	CP0{.b}	f			
Operands:	f ∈ [0 ... 8191]					
Operation:	(f) - 0x0000					
Status Affected:	C, DC, N, OV, Z					
Encoding:	1110	0010	0B0f	ffff	ffff	ffff
Description:	<p>Compute (f) - 0x0000, set flags but do not store result.</p> <p>The 'B' bit selects byte or word operation.</p> <p>The 'f' bits select the address of the file register.</p> <p>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</p>					
Words:	1					
Cycles:	1					
Example:		CP0	53	: Compare		
	Before Instruction					
	After Instruction					

TABLE 65 -- CPFB: Compare f with Ww with Borrow, Set status flags

Syntax:	{label:}	CPB{.b}	f			
Operands:	<u>f ∈ [0 ...8191]</u>					
Operation:	<u>(f) - (Ww) - (C)</u>					
Status Affected:	<u>C, DC, N, OV, Z</u>					
Encoding:	<u>1110</u>	<u>0011</u>	<u>1B0f</u>	<u>ffff</u>	<u>ffff</u>	<u>ffff</u>
Description:	<u>Compute (f) - (Ww) - (C), set C flags but do not store result.</u> <u>The 'B' bit selects byte or word operation.</u> <u>The 'f' bits select the address of the file register.</u> <u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>CPB</u>	<u>RAM135</u>	<u>; Compare RAM135-Ww</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 66 -- CPLS: Compare Wb with lit5, Set status flags

Syntax:	{label:}	CP{.b}	Wb,	lit5		
Operands:	Wb \in [W0 ... W15]; lit5 \in [0 ... 31]					
Operation:	(Wb) - lit5					
Status Affected:	C, DC, N, OV, Z					
Encoding:	1110	0001	0www	wB00	011k	kkkk
Description:	<p><u>Compute (Wb) - lit5, set flags but do not store result.</u></p> <p><u>The 'B' bit selects byte or word operation.</u></p> <p><u>The 'w' bits select the address of the Wb base register.</u></p> <p><u>The 'k' bits provide the literal operand, a five bit integer number.</u></p> <p><u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
Words:	1					
Cycles:	1					
Example:		CP	W5, #30			
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 67 -- DAW: Decimal Adjust Wn

<u>Syntax:</u>	<u>{label:}</u>	<u>DAW.b</u>	<u>Wn</u>			
<u>Operands:</u>	<u>Wn</u> ∈ [W0 ... W15]					
<u>Operation:</u>	<u>If [Wn<3:0> >9] or [DC = 1] then</u> <u>(Wn<3:0>) + 6 → Wn<3:0></u> <u>else</u> <u>(Wn<3:0>) → Wn<3:0>;</u> <u>If [Wn<7:4> >9] or [C = 1] then</u> <u>(Wn<7:4>) + 6 → Wn<7:4></u> <u>else</u> <u>(Wn<7:4>) → Wn<7:4>;</u>					
<u>Status Affected:</u>	<u>C</u>					
<u>Encoding:</u>	<u>1111</u>	<u>1101</u>	<u>0100</u>	<u>0000</u>	<u>0000</u>	<u>ssss</u>
<u>Description:</u>	<u>DAW adjusts the eight bit value in Wn (LSB's) resulting from the earlier addition of two variables (each in packed BCD format) and produces a correct packed BCD result.</u> <u>The 's' bits select the address of the source register.</u> <u>This instruction operates in byte mode only.</u> <u>The .b extension must be included with the opcode.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1</u>					
<u>Example:</u>		<u>DAW.b</u>	<u>W5</u>	<u>: Decimal adjust</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 68 -- DEC: Decrement Ws

Syntax:	{label:}	DEC{.b}	Ws,	Wd		
			[Ws],	[Wd]		
			[Ws]++,	[Wd]++		
			[Ws]--,	[Wd]--		
			[Ws++] ,	[Wd++]		
			[Ws--] ,	[Wd--]		
Operands:	Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]					
Operation:	(Ws) - 1 → Wd					
Status Affected:	C, DC, N, OV, Z					
Encoding:	1110	1001	0Bqq	qddd	dppp	ssss
Description:	<p><u>Subtract one from the contents of the source register Ws and place the result in the destination register Wd.</u></p> <p><u>The 'B' bit selects byte or word operation.</u></p> <p><u>The 's' bits select the address of the source register.</u></p> <p><u>The 'd' bits select the address of the destination register.</u></p> <p><u>The 'p' bits select the source address mode 2 (values 0-4).</u></p> <p><u>The 'q' bits select the destination address mode 2 (values 0-4).</u></p> <p><u>See Table 1-5 and Table 1-6 for modifier addressing information.</u></p> <p><u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
Words:	1					
Cycles:	1					
Example:		DEC	W5,W7	: Decrement		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 69 -- DEC2: Decrement Ws by 2

Syntax:	{label:}	DEC2{.b}	Ws,	Wd		
			[Ws],	[Wd]		
			[Ws]++,	[Wd]++		
			[Ws]--,	[Wd]--		
			[Ws++] ,	[Wd++]		
			[Ws--] ,	[Wd--]		
Operands:	Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]					
Operation:	(Ws) - 2 → Wd					
Status Affected:	C, DC, N, OV, Z					
Encoding:	1110	1001	1Bqq	qddd	dppp	ssss
Description:	<p>Subtract two from the contents of the source register Ws and place the result in the destination register Wd.</p> <p>The 'B' bit selects byte or word operation.</p> <p>The 's' bits select the address of the source register.</p> <p>The 'd' bits select the address of the destination register.</p> <p>The 'p' bits select the source address mode 2 (values 0-4).</p> <p>The 'q' bits select the destination address mode 2 (values 0-4).</p> <p>See Table 1-5 and Table 1-6 for modifier addressing information.</p> <p>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</p>					
Words:	1					
Cycles:	1					
Example:		DEC2	W5,W7	; Decrement		
	Before Instruction					
	After Instruction					

TABLE 70 -- DECF: Decrement f

Syntax:	{label:}	DEC{.b}	f	{,Ww}		
Operands:	$f \in [0 \dots 8191]$					
Operation:	$(f) - 1 \rightarrow \text{destination designated by D}$					
Status Affected:	C, DC, N, OV, Z					
Encoding:	1110	1101	0Bdf	ffff	ffff	ffff
Description:	<p><u>Subtract one from the contents of the file register and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.</u></p> <p><u>The 'B' bit selects byte or word operation.</u></p> <p><u>The 'f' bits select the address of the file register.</u></p> <p><u>The 'D' bit selects the destination.</u></p> <p><u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
Words:	1					
Cycles:	1					
Example:		DECF	RAM135	: Decrement		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 71 -- DISI: Disable Interrupts

Syntax:	{label:}	DISI	lit14			
Operands:	lit14 ∈ [0 ... 16384]					
Operation:	Disable interrupts for lit14 cycles					
Status Affected:	None					
Encoding:	1111	1100	00kk	kkkk	kkkk	kkkk
Description:	<u>This instruction disables the interrupts for lit14 instruction cycles after the instruction executes. This instruction can be used before critical code sections to ensure un-interrupted execution.</u>					
Words:	1					
Cycles:	1					
Example:		DISI	#30	<u>: Disable interrupts for next 30 instruction cycles</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 72 -- DO: Initialize Hardware loop

Syntax:	{label:}	DO	Slit16,	lit14		
Operands:	Slit16 $\in [-32768 \dots +32767]$; lit14 $\in [0 \dots 16383]$					
Operation:	Push Shadows (lit14) \rightarrow DOCOUNT (Loop Count Register) (PC)+4 \rightarrow PC (PC) \rightarrow DOSTART (Loop Start Register) (PC) + (2*Slit16) \rightarrow DOEND (Loop End Register) Enable Code Looping					
Status Affected:	None					
Encoding:	<u>0000</u>	<u>1000</u>	<u>00kk</u>	<u>kkkk</u>	<u>kkkk</u>	<u>kkkk</u>
	<u>0000</u>	<u>0000</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>
Description:	Repeat lit14 times the code segment delineated by the address of the instruction immediately following the DO instruction and an end address formed by the address of the first instruction plus offset Slit16. The 'k' bits specify the loop count. The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+4) of the last instruction executed in the loop. Note 1: The value k = 0 is invalid. 2: The value n=-1 is invalid. The DO instruction is not allowed to generate a DO loop only including itself. 3: n=0 will generate a loop size of 1 word (same as REPEAT instruction except instruction is fetched every iteration).					
Words:	2					
Cycles:	2 + n*(# of cycles required to execute loop)					
Example:		DO	#5, #6	; Do next 5 instructions 6 times		
	Before Instruction					
	After Instruction					

TABLE 73 -- DOW: Initialize Hardware loop

<u>Syntax:</u>	<u>{label:}</u>	<u>DO</u>	<u>Slit16,</u>	<u>Wn</u>		
<u>Operands:</u>	<u>Slit16 ∈ [-32768 ... +32767];</u> <u>Wn ∈ [W0 ... W15]</u>					
<u>Operation:</u>	<u>Push Shadows</u> <u>(Wn) → DOCOUNT (Loop Count Register)</u> <u>(PC)+4 → PC</u> <u>(PC) → DOSTART (Loop Start Register)</u> <u>(PC) + (2*Slit16) → DOEND (Loop End Register)</u> <u>Enable Code Looping</u>					
<u>Status Affected:</u>	<u>None</u>					
	<u>0000</u>	<u>1000</u>	<u>1000</u>	<u>0000</u>	<u>0000</u>	<u>ssss</u>
<u>Encoding:</u>	<u>0000</u>	<u>0000</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>
<u>Description:</u>	<u>Repeat (Wn) times the code segment delineated by the address of the instruction immediately following the DO instruction and an end address formed by the address of the first instruction plus offset Slit16.</u> <u>The 's' bits specify the register Wn that contains the loop count (only the 14 LSBs of (Wn) are considered).</u> <u>The 'n' bits are a signed literal that specifies the number of instructions offset from (PC+4) of the last instruction executed in the loop.</u> <u>Note 1: The value (Wn) = 0 is invalid.</u> <u>2: The value n=-1 is invalid. The DO instruction is not allowed to generate a DO loop only including itself.</u> <u>3: n=0 will generate a loop size of 1 word (same as REPEAT instruction except instruction is fetched every iteration).</u>					
<u>Words:</u>	<u>2</u>					
<u>Cycles:</u>	<u>2 + n*(# of cycles required to execute loop)</u>					
<u>Example:</u>		<u>DO</u>	<u>#5,W6</u>	<u>: Do next 5 instructions (W6) times</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 74 -- ED: Euclidean Distance

Syntax:	{label:}ED	A,	Wm*Wm	,Wxp,[Wx]	,[Wy]	
		B,		,Wxp,[Wx]+=kx	,[Wy]+=ky	
				,Wxp,[Wx]-=kx ‡	,[Wy]-=ky ‡	
				,Wxp,[W5+W8]	,[W7+W8]	
				none	none	
				‡ Alternate format for negative kx,ky		
Operands:	Wm*Wm ∈ {W0*W0; W1*W1; W2*W2; W3*W3} Wxp ∈ {W0 ... W3}; Wx ∈ {W4, W5}; kx ∈ {-6, -4, -2, 2, 4, 6}; Wy ∈ {W6, W7}; ky ∈ {-6, -4, -2, 2, 4, 6};					
Operation:	(Wm)*(Wm) → ACC(A or B); ([Wx]-{Wy]) → Wxp; (Wx)+kx → Wx; (Wy)+ky → Wy;					
Status Affected:	OA, OB, SA, SB					
Encoding:	1111	00mm	A1xx	00ii	ijjj	jj11
Description:	<p>Instruction to compute (A-B)² functions. Prefetch computes difference of prefetched values. Then, the Wm register is squared. The 32-bit result is sign-extended to 40-bits and written to the specified accumulator.</p> <p>Wx register specifies the prefetch of the minuend register. The prefetch is done with indirect, indirect with post inc/dec, indirect with register offset, copy of the other prefetch or none. Post-modify Wx as required. Wy register specifies the prefetch of the subtrahend register. Post-modify Wy as required. Wxp contains the difference result.</p> <p>The 'm' bits select the operand register Wm for the square: The 'A' bit selects the accumulator for the result. The 'i' bits select the Wx pre-fetch operation. The 'j' bits select the Wy pre-fetch operation. The 'x' bits select the pre-fetch difference Wxp destination.</p> <p>See Table 1-9 through Table 1-14 for modifier addressing information.</p>					
Words:	1					
Cycles:	1					
Example:	ED	A,W2*W2,W0,[W4]-=6,[W6]			; Euclidean Distance to ACCA	
	Before Instruction					
	ACCA = 2					
	ACCB = 3					
	W0 = 5					
	W1 = 6					
	W2 = 7					

	<u>W3 = 8</u> <u>W8 = 1000</u> <u>W10 = 2000</u> <u>RAM(994) = 16</u> <u>RAM(1000) = 17</u> <u>RAM(2000) = 18</u> <u>After Instruction</u> <u>ACCA = 2+7*8=58</u> <u>ACCB = 3</u> <u>W0 = 17</u> <u>W1 = 18</u> <u>W2 = 7</u> <u>W3 = 8</u> <u>W8 = 994</u> <u>W10 = 2000</u> <u>RAM(994) = 3</u> <u>RAM(1000) = 17</u> <u>RAM(2000) = 18</u>	
--	--	--

TABLE 75 -- EXCH: Exchange Ws and Wd

<u>Syntax:</u>	<u>{label:}</u>	<u>EXCH</u>	<u>Wns,</u>	<u>Wnd</u>		
<u>Operands:</u>	<u>Wns ∈ [W0 ... W15]; Wnd ∈ [W0 ... W15]</u>					
<u>Operation:</u>	<u>(Wns) ↔ (Wnd)</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>1111</u>	<u>1101</u>	<u>0000</u>	<u>0ddd</u>	<u>d000</u>	<u>ssss</u>
<u>Description:</u>	<u>This instruction exchanges the contents of two working registers.</u> <u>The 's' bits select the address of one of the registers.</u> <u>The 'd' bits select the address of the other register.</u> <u>Note: Word operation is assumed.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1</u>					
<u>Example:</u>		<u>EXCH</u>	<u>W5,W6</u>	<u>: Exchange W5 and W6</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 76 -- GOTO: Unconditional Branch

Syntax:	{label:}	GOTO	lit23			
Operands:	lit23 ∈ [0 ... 8388606]					
Operation:	lit23 → PC, NOP → Instruction Register.					
Status Affected:	None					
Encoding:						
1st word	0000	0100	nnnn	nnnn	nnnn	nnn0
2nd word	0000	0000	0000	0000	0nnn	nnnn
Description:	Unconditional branch to anywhere within the 4M instruction program memory range. GOTO is always a two-cycle instruction. The 'n' bits form the target address.					
Words:	2					
Cycles:	2					
Example:		GOTO	label	: Goto location at label		
	Before Instruction					
	After Instruction					

TABLE 77 -- GOTOW: Unconditional Indirect Branch

<u>Syntax:</u>	<u>{label:}</u>	<u>GOTO</u>	<u>Wn</u>			
<u>Operands:</u>	<u>Wn ∈ [W0 ... W15]</u>					
<u>Operation:</u>	<u>0 → PC<22:17>, (Wn) → PC<16:1>, 0 → PC<0>;</u> <u>NOP → Instruction Register.</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>0000</u>	<u>0001</u>	<u>0100</u>	<u>0000</u>	<u>0000</u>	<u>ssss</u>
<u>Description:</u>	<u>Unconditional indirect branch within the first 64K instructions program</u> <u>memory range. GOTO is always a two-cycle instruction.</u> <u>The 16-bit value (Wn) is left shifted 1 bit, zero-extended and loaded into the</u> <u>PC. CALL is a two-cycle instruction.</u> <u>The 's' bits select the address of the source register.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>2</u>					
<u>Example:</u>		<u>GOTO</u>	<u>W5</u>	<u>; Goto location specified by contents of W5</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 78 -- INC: Increment Ws

Syntax:	{label:}	INC{.b}	Ws,	Wd		
			<u>[Ws],</u>	<u>[Wd]</u>		
			<u>[Ws]++</u> ,	<u>[Wd]++</u>		
			<u>[Ws]--</u> ,	<u>[Wd]--</u>		
			<u>[Ws++]</u> ,	<u>[Wd++]</u>		
			<u>[Ws--]</u> ,	<u>[Wd--]</u>		
Operands:	Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]					
Operation:	(Ws) + 1 → Wd					
Status Affected:	C, DC, N, OV, Z					
Encoding:	<u>1110</u>	<u>1000</u>	<u>0Bqq</u>	<u>qddd</u>	<u>dppp</u>	<u>ssss</u>
Description:	<p>Add one to the contents of the source register Ws and place the result in the destination register Wd.</p> <p>The 'B' bit selects byte or word operation.</p> <p>The 's' bits select the address of the source register.</p> <p>The 'd' bits select the address of the destination register.</p> <p>The 'p' bits select the source address mode 2 (values 0-4).</p> <p>The 'q' bits select the destination address mode 2 (values 0-4).</p> <p>See Table 1-5 and Table 1-6 for modifier addressing information.</p> <p>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</p>					
Words:	1					
Cycles:	1					
Example:		INC	W5,W7	; Increment		
	Before Instruction					
	After Instruction					

TABLE 79 -- INC2: Increment Ws by 2

Syntax:	{label:}	INC2	Ws,	Wd		
			[Ws],	[Wd]		
			[Ws]++,	[Wd]++		
			[Ws]--,	[Wd]--		
			[Ws++] ,	[Wd++]		
			[Ws--] ,	[Wd--]		
Operands:	Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]					
Operation:	(Ws) + 2 → Wd					
Status Affected:	C, DC, N, OV, Z					
Encoding:	1110	1000	1Bqq	qddd	dppp	ssss
Description:	<p>Add two to the contents of the source register Ws and place the result in the destination register Wd.</p> <p>The 'B' bit selects byte or word operation.</p> <p>The 's' bits select the address of the source register.</p> <p>The 'd' bits select the address of the destination register.</p> <p>The 'p' bits select the source address mode 2 (values 0-4).</p> <p>The 'q' bits select the destination address mode 2 (values 0-4).</p> <p>See Table 1-5 and Table 1-6 for modifier addressing information.</p> <p>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</p>					
Words:	1					
Cycles:	1					
Example:		INC	W5,W7	: Increment		
	Before Instruction					
	After Instruction					

TABLE 80 -- INCF: Increment f

Syntax:	{label:}	INC{.b}	f	{,Ww}		
Operands:	f ∈ [0 ... 8191]					
Operation:	(f) + 1 → destination designated by D					
Status Affected:	C, DC, N, OV, Z					
Encoding:	1110	1100	0BDf	ffff	ffff	ffff
Description:	<p>Add one to the contents of the file register and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.</p> <p>The 'B' bit selects byte or word operation.</p> <p>The 'f' bits select the address of the file register.</p> <p>The 'D' bit selects the destination.</p> <p>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</p>					
Words:	1					
Cycles:	1					
Example:		INC		RAM135		: Increment
	Before Instruction					
	After Instruction					

TABLE 81 -- IOR: Inclusive Or Wb and Ws

Syntax:	{label:}	IOR{.b}	Wb,	Ws,	Wd	
				[Ws],	[Wd]	
				[Ws]++,	[Wd]++	
				[Ws]--,	[Wd]--	
				[Ws++] ,	[Wd++]	
				[Ws--] ,	[Wd--]	
Operands:	Wb ∈ [W0 ... W15]; Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]					
Operation:	(Wb).IOR.(Ws) → Wd					
Status Affected:	N, Z					
Encoding:	0111	0www	wBqq	qddd	dppp	ssss
Description:	<p>for the contents of the source register Ws and the contents of the base register Wb and place the result in the destination register Wd.</p> <p>The 'B' bit selects byte or word operation.</p> <p>The 's' bits select the address of the source register.</p> <p>The 'w' bits select the address of the base register.</p> <p>The 'd' bits select the address of the destination register.</p> <p>The 'p' bits select source address mode 2.</p> <p>The 'q' bits select destination address mode 2.</p> <p>See Table 1-5 and Table 1-6 for modifier addressing information.</p> <p>The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</p>					
Words:	1					
Cycles:	1					
Example:		IOR	W5,W6,W7	; Inclusive Or		
	Before Instruction					
	After Instruction					

TABLE 82 -- IORLS: Inclusive Or Wb and Short Literal

Syntax:	<u>{label:}</u>	<u>IOR(.b)</u>	<u>Wb</u>	<u>lit5</u>	<u>Wd</u>	
					<u>[Wd]</u>	
					<u>[Wd]++</u>	
					<u>[Wd]--</u>	
					<u>[Wd++]</u>	
					<u>[Wd--]</u>	
Operands:	<u>Wb ∈ [W0 ... W15]; lit5 ∈ [0 ... 31]; Wd ∈ [W0 ... W15]</u>					
Operation:	<u>(Wb).IOR.lit5 → Wd</u>					
Status Affected:	<u>N, Z</u>					
Encoding:	<u>0111</u>	<u>0www</u>	<u>wBqq</u>	<u>qddd</u>	<u>d11k</u>	<u>kkkk</u>
Description:	<p><u>Compute the Inclusive Or of the contents of the base register Wb and the literal operand and place the result in the destination register Wd.</u></p> <p><u>The 'B' bit selects byte or word operation.</u></p> <p><u>The 'w' bits select the address of the base register.</u></p> <p><u>The 'k' bits provide the literal operand, a five-bit integer number.</u></p> <p><u>The 'd' bits select the address of the destination register.</u></p> <p><u>The 'q' bits select destination address mode 2.</u></p> <p><u>See Table 1-6 for modifier addressing information.</u></p> <p><u>The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>IOR</u>		<u>W5,#12,W7</u>	<u>; Add</u>	
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 83 -- IORLW: Inclusive Or Literal and Wn

Syntax:	{label:}	IOR{.b}	Slit10,	Wn		
Operands:	Slit10 \in [-512 ... 511]; Wn \in [W0 ... W15]					
Operation:	Slit10.IOR.(Wn) \rightarrow Wn					
Status Affected:	N, Z					
Encoding:	1011	0011	0Bkk	kkkk	kkkk	dddd
Description:	<p>Compute the Inclusive Or of the literal operand and the contents of the working register Wn and place the result in the working register Wn.</p> <p>The 'B' bit selects byte or word operation.</p> <p>The 'd' bits select the address of the working register.</p> <p>The 'k' bits specify the literal operand, a signed 10-bit number.</p> <p>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</p>					
Words:	1					
Cycles:	1					
Example:		IOR	#123,W7	: Inclusive Or		
	Before Instruction					
	After Instruction					

TABLE 84 -- IORWF: Inclusive Or f and Ww

	<u>{label:}</u>	<u>IOR{.b}</u>	<u>f</u>	<u>{.Ww}</u>		
<u>Operands:</u>	<u>f ∈ [0 ... 8191]</u>					
<u>Operation:</u>	<u>(f).IOR(.Ww) → destination designated by D</u>					
<u>Status Affected:</u>	<u>N, Z</u>					
<u>Encoding:</u>	<u>1011</u>	<u>0111</u>	<u>0B D f</u>	<u>f f f f</u>	<u>f f f f</u>	<u>f f f f</u>
<u>Description:</u>	<p><u>Compute the IOR of the contents of the working register and the contents of the file register and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.</u></p> <p><u>The 'B' bit selects byte or word operation.</u></p> <p><u>The 'D' bit selects the destination.</u></p> <p><u>The 'f' bits select the address of the file register.</u></p> <p><u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1</u>					
<u>Example:</u>		<u>IOR</u>		<u>RAM135, Ww</u>	<u>; Inclusive Or</u>	
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 85 -- ITCH: Pop Shadow Registers

Syntax:	{label:}	POP.S				
Operands:	None					
Operation:	Pop shadow registers					
Status Affected:	All					
Encoding:	<u>1111</u>	<u>1110</u>	<u>1000</u>	<u>0000</u>	<u>0000</u>	<u>0000</u>
Description:	<u>The values in the shadow registers are copied into the primary registers.</u>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>ITCH</u>			<u>: Itch</u>	
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 86 -- LAC: Load Accumulator A

Syntax:	{label:}	LAC	A, B,	Wns, [Wns], [Wns]++ [Wns]-- [Wns--], [Wns+Wb], [Wns+lit5]	[, Slit4]	
Operands:	Wns ∈ [W0 ... W15]; Wb ∈ [W0 ... W15]; lit5 ∈ [0... 31] Slit4 ∈ [-8 ... +7]					
Operation:	ShiftSlit4(Extend(Wns)) → AXX					
Status Affected:	None					
Encoding:	1100	1010	Awww	wrrr	rggg	ssss
Description:	<p><u>Read the contents of the effective address. Optionally shift, then place result in accumulator.</u></p> <p><u>The value contained at the effective address is assumed to be Q15 fractional data and is automatically sign-extended (through bit 39) and zero-backfilled (bits [15:0]) prior to shifting.</u></p> <p><u>The 'A' bits specify the destination accumulator.</u> <u>The 's' bits specify the source register Wns.</u> <u>The 'g' bits select source address mode 3.</u> <u>The 'w' bits specify the offset amount lit5 OR the offset register Wb.</u> <u>The 'r' bits encode the optional operand Slit4 which determines the amount of the accumulator preshift; if the operand Slit4 is absent, a 0 is encoded.</u></p> <p><u>See Table 1-7 for modifier addressing information.</u></p> <p><u>Note: Positive values of operand Slit4 represent arithmetic shift right.</u> <u>Negative values of operand Slit4 represent shift left.</u></p>					
Words:	1					
Cycles:	1					
Example:		LAC	A,W5		; Load Accumulator A	
	Before Instruction					
	After Instruction					

TABLE 87 -- LNK: Allocate Stack Frame

Syntax:	{label:}	LNK	lit14			
Operands:	lit14 [0 ... 16384]					
Operation:	(W14) → [W15]--; (W15) → W14; (W15) - lit14 → W15					
Status Affected:	None					
Encoding:	1111	1010	00kk	kkkk	kkkk	kkkk
Description:	<u>This instruction allocates a stack frame of size lit14 and adjusts the stack pointer and frame pointer.</u> <u>The 'k' bits specify the size of the stack frame.</u>					
Words:	1					
Cycles:	1					

TABLE 88 -- LSR: Logical Shift Right Ws

Syntax:	<u>{label:}</u>	<u>LSR{.b}</u>	<u>Ws,</u>	<u>Wd</u>		
			<u>[Ws],</u>	<u>[Wd]</u>		
			<u>[Ws]++,</u>	<u>[Wd]++</u>		
			<u>[Ws]--,</u>	<u>[Wd]--</u>		
			<u>[Ws++]</u> ,	<u>[Wd++]</u>		
			<u>[Ws--]</u> ,	<u>[Wd--]</u>		
Operands:	<u>Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]</u>					
Operation:	<u>For word operation:</u> <u>0 → Wd<15>, (Ws<15:1>) → Wd<14:0>, (Ws<0>) → C</u> <u>For byte operation:</u> <u>0 → Wd<7>, (Ws<7:1>) → Wd<6:0>, (Ws<0>) → C</u> <div style="text-align: center;"> </div>					
Status Affected:	<u>C, N, OV, Z</u>					
Encoding:	<u>1101</u>	<u>0001</u>	<u>0Bqq</u>	<u>qddd</u>	<u>dppp</u>	<u>ssss</u>
Description:	<u>Shift the contents of the source register Ws one bit to the right and place the result in the destination register Wd. The Carry Flag bit is set if the LSB of Ws is '1'.</u> <u>The 'B' bit selects byte or word operation.</u> <u>The 's' bits select the address of the source register.</u> <u>The 'd' bits select the address of the destination register.</u> <u>The 'p' bits select source address mode 2.</u> <u>The 'q' bits select destination address mode 2.</u> <u>See Table 1-5 and Table 1-6 for modifier addressing information.</u> <u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>LSR</u>	<u>W5,W6</u>	<u>: Shift right</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 89 -- LSRF: Logical Shift Right f


Syntax:	<u>{label:}</u>	<u>LSR{.b}</u>	<u>f</u>	<u>{,Ww}</u>		
Operands:	<u>f ∈ [0 ... 8191]</u>					
Operation:	<u>For word operation:</u> <u>0 → Dest<15>, (f<15:1>) → Dest<14:0>, (f<0>) → C</u> <u>For byte operation:</u> <u>0 → Dest<7>, (f<7:1>) → Dest<6:0>, (f<0>) → C</u> 					
Status Affected:	<u>C, N, OV, Z</u>					
Encoding:	<u>1101</u>	<u>0101</u>	<u>0BDf</u>	<u>ffff</u>	<u>ffff</u>	<u>ffff</u>
Description:	<u>Shift the contents of the file register f one bit to the right and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register. The carry flag bit is set if the LSB of the file register is '1'.</u> <u>The 'B' bit selects byte or word operation.</u> <u>The 'D' bit selects the destination.</u> <u>The 's' bits select the address of the working register.</u> <u>The 'f' bits select the address of the file register.</u> <u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>LSR</u>		<u>RAM135, Ww</u>		<u>: Shift right</u>
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 90 -- MAC: Multiply and Accumulate

Syntax:	<u>{label:}MAC</u>	<u>A</u> <u>B</u>	<u>Wm*Wn</u>	<u>,Wxp,[Wx]</u> <u>,Wxp,[Wx]+=kx</u> <u>,Wxp,[Wx]-=kx ‡</u> <u>,Wxp,[W5+W8]</u> <u>none</u>	<u>,Wyp,[Wy]</u> <u>,Wyp,[Wy]+=ky</u> <u>,Wyp,[Wy]-=ky ‡</u> <u>,Wyp,[W7+W8]</u> <u>none</u>	<u>,AWB</u> <u>none</u>
				<u>‡ Alternate format for negative kx,ky</u>		
Operands:	<u>Wm*Wn ∈ {W0*W1; W0*W2; W0*W3; W1*W2; W1*W3; W2*W3}</u> <u>Wxp ∈ {W0 ... W3}; Wx ∈ {W4, W5}; kx ∈ {-6, -4, -2, 2, 4, 6};</u> <u>Wyp ∈ {W0 ... W3}; Wy ∈ {W6, W7}; ky ∈ {-6, -4, -2, 2, 4, 6};</u> <u>AWB ∈ {W9, [W9]++}</u>					
Operation:	<u>(ACC(A or B)) + (Wm)*(Wn) → ACC(A or B);</u> <u>([Wx]) → Wxp; (Wx)+kx→Wx;</u> <u>([Wy]) → Wyp; (Wy)+ky→Wy;</u> <u>(ACC(B or A)) rounded → AWB</u>					
Status Affected:	<u>OA, OB, SA, SB</u>					
Encoding:	<u>1100</u>	<u>Ommm</u>	<u>A0xx</u>	<u>yyii</u>	<u>ijjj</u>	<u>jja a</u>
Description:	<u>Signed, fractional or integer multiply the contents of two W registers. The 32-bit result is sign-extended to 40-bits and added to the specified accumulator.</u> <u>Wx register specifies the prefetch of the multiplier Wxp register. The prefetch is done with indirect, indirect with post inc/dec, indirect with register offset, copy of the other prefetch or none. Post-modify Wx as required.</u> <u>Wy register specifies the prefetch of the multiplier Wyp register. Post-modify Wy as required.</u> <u>AWB specifies the direct or indirect store of the convergently rounded contents of other accumulator, if required.</u> <u>The 'm' bits select the operand registers Wm and Wn for the multiply.</u> <u>The 'A' bit selects the accumulator for the result. The other accumulator is used for write back.</u> <u>The 'i' bits select the Wx pre-fetch operation.</u> <u>The 'j' bits select the Wy pre-fetch operation.</u> <u>The 'x' bits select the pre-fetch Wxp destination.</u> <u>The 'y' bits select the pre-fetch Wyp destination.</u> <u>The 'a' bits select the accumulator write-back destination.</u> <u>See Table 1-9 through Table 1-14 for modifier addressing information.</u>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:	<u>MAC</u>	<u>A,W2*W3,W0=[W4]-=6,W1=[W6],[W9]++</u>			<u>; Multiply and Accumulate A</u>	

	<p><u>Before Instruction</u></p> <p><u>ACCA = 2</u></p> <p><u>ACCB = 3</u></p> <p><u>W0 = 5</u></p> <p><u>W1 = 6</u></p> <p><u>W2 = 7</u></p> <p><u>W3 = 8</u></p> <p><u>W8 = 1000</u></p> <p><u>W10 = 2000</u></p> <p><u>RAM(994) = 16</u></p> <p><u>RAM(1000) = 17</u></p> <p><u>RAM(2000) = 18</u></p> <p><u>After Instruction</u></p> <p><u>ACCA = 2+7*8=58</u></p> <p><u>ACCB = 3</u></p> <p><u>W0 = 17</u></p> <p><u>W1 = 18</u></p> <p><u>W2 = 7</u></p> <p><u>W3 = 8</u></p> <p><u>W8 = 994</u></p> <p><u>W10 = 2000</u></p> <p><u>RAM(994) = 3</u></p> <p><u>RAM(1000) = 17</u></p> <p><u>RAM(2000) = 18</u></p>	
--	---	--

TABLE 91 -- MOV: Move Ws to Wd

Syntax:	{label:}	MOV{.b}	Ws,	Wd		
			[Ws],	[Wd]		
			[Ws]++	[Wd]++		
			[Ws]--	[Wd]--		
			[Ws--],	[Wd--]		
			[Ws+Wb],	[Wd+Wb]		
			[Ws+lit5],	[Wd+lit5]		
Operands:	Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]; Wb ∈ [W0 ... W15]; lit5 ∈ ... 31]					
Operation:	(EAs) → EAd					
Status Affected:	None					
Encoding:	0111	1www	wBhh	hddd	dggg	ssss
Description:	<p>Move the contents of the source register into the destination register.</p> <p>The 'B' bit selects byte or word operation.</p> <p>The 's' bits select the address of the source register.</p> <p>The 'd' bits select the address of the destination register.</p> <p>The 'g' bits select source address mode 3.</p> <p>The 'h' bits select destination address mode 3.</p> <p>The 'w' bits define the addressing mode literal 'lit5' or offset Wb; these bits are shared by source and destination addresses.</p> <p>See Table 1-7 and Table 1-8 for modifier addressing information.</p> <p>The assembly mnemonics PUSH Ws and POP Wd translate to MOV.</p> <p>Note: The extension .b in the instruction denotes a byte move rather than a word move. You may use a .w extension to denote a word move, but it is not required.</p>					
Words:	1					
Cycles:	1					
Example:		MOV	W5,W6	; Move W5 to W6		
	Before Instruction					
	After Instruction					

TABLE 92 -- MOVE: Move f to destination

<u>Syntax:</u>	<u>{label:}</u>	<u>MOV{.b}</u>	<u>f</u>	<u>{,Ww}</u>		
<u>Operands:</u>	<u>f ∈ [0 ... 8191];</u>					
<u>Operation:</u>	<u>(f) → destination designated by D</u>					
<u>Status Affected:</u>	<u>Z, N</u>					
<u>Encoding:</u>	<u>1011</u>	<u>1111</u>	<u>1BDf</u>	<u>ffff</u>	<u>ffff</u>	<u>ffff</u>
<u>Description:</u>	<u>Move the contents of the file register to the destination designated by D: if D=0, put the value into Ww, if D=1 the only effect is to modify the status flags, no writeback is required.</u> <u>The 'B' bit selects byte or word operation.</u> <u>The 'D' bit selects the destination, (0 for Wd, 1 for f).</u> <u>The 'f' bits select the address of the file register.</u> <u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1</u>					
<u>Example:</u>		<u>MOV</u>	<u>RAM433, Ww</u>	<u>: Move File register 433 to Ww</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 93 -- MOVL: Move 16-bit literal to Wd

<u>Syntax:</u>	<u>{label:}</u>	<u>MOV</u>	<u>lit16,</u>	<u>Wn</u>		
<u>Operands:</u>	<u>lit16 ∈ [-32768 ... 65535]; Wn ∈ [W0 ... W15]</u>					
<u>Operation:</u>	<u>lit16 → Wn</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>0010</u>	<u>dddd</u>	<u>kkkk</u>	<u>kkkk</u>	<u>kkkk</u>	<u>kkkk</u>
<u>Description:</u>	<u>The Literal 'k' is loaded into Wn register.</u> <u>The 'd' bits select the address of the working register.</u> <u>The 'k' bits specify the value of the literal.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1</u>					
<u>Example:</u>		<u>MOV</u>	<u>#64159, W5</u>	<u>: Move 64159 into W5</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 94 -- MOV SAC: Prefetch Operands and Store Accumulator

Syntax:	{label:}	MOV SAC	A.	Wxp,[Wx]	Wyp,[Wy]	AWB
			B.	Wxp,[Wx]+=kx	Wyp,[Wy]+=ky	none
				Wxp,[Wx]-=kx ‡	Wyp,[Wy]-=ky ‡	
				Wxp,[W5+W8]	Wyp,[W7+W8]	
				none	none	
				‡ Alternate format for negative kx,ky		
Operands:	Wxp ∈ {W0 ... W3}; Wx ∈ {W4, W5}; kx ∈ {-6, -4, -2, 2, 4, 6}; Wyp ∈ {W0 ... W3}; Wy ∈ {W6, W7}; ky ∈ {-6, -4, -2, 2, 4, 6}; AWB ∈ {W9, [W9]++}					
Operation:	([Wx]) → Wxp; (Wx)+kx→Wx; ([Wy]) → Wyp; (Wy)+ky→Wy; (ACC(B or A)) rounded → AWB					
Status Affected:	OA, OB, SA, SB					
Encoding:	1100	0111	A0xx	yyii	Iiij	jjaa
Description:	<p>Prefetch operands and optionally store accumulator results in preparation for a repeated MAC type instruction.</p> <p>Wx register specifies the prefetch of the multiplier Wxp register. The prefetch is done with indirect, indirect with post inc/dec, indirect with register offset, copy of the other prefetch or none. Post-modify Wx as required.</p> <p>Wy register specifies the prefetch of the multiplier Wyp register. Post-modify Wy as required.</p> <p>AWB specifies the direct or indirect store of the convergently rounded contents of other accumulator, if required. Note that the specification of (B or A) is consistent with the MAC instruction. For example, MOV SAC A,W9 will store ACCB into W9.</p> <p>The 'A' bit selects the other accumulator used for write back.</p> <p>The 'i' bits select the Wx pre-fetch operation.</p> <p>The 'j' bits select the Wy pre-fetch operation.</p> <p>The 'x' bits select the pre-fetch Wxp destination.</p> <p>The 'y' bits select the pre-fetch Wyp destination.</p> <p>The 'a' bits select the accumulator write-back destination.</p> <p>See Table 1-9 through Table 1-14 for modifier addressing information.</p>					
Words:	1					
Cycles:	1					
Example:	MOV SAC A,W0,[W4]-=6,W1,[W6],W9 ; Prefetch and move ACCB to W9					
	Before Instruction					
	ACCA = 2					
	ACCB = 3					
	W0 = 5					
	W1 = 6					

	<u>W2 = 7</u> <u>W3 = 8</u> <u>W8 = 1000</u> <u>W10 = 2000</u> <u>RAM(994) = 16</u> <u>RAM(1000) = 17</u> <u>RAM(2000) = 18</u> <u>After Instruction</u>	
--	---	--

TABLE 95 -- MOVWF: Move Ww to F

<u>Syntax:</u>	<u>{label:}</u>	<u>MOV{.b}</u>	<u>Ww,</u>	<u>f</u>		
<u>Operands:</u>	<u>f ∈ [0 ... 8191]</u>					
<u>Operation:</u>	<u>(Ww) → f</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>1011</u>	<u>0111</u>	<u>1B1f</u>	<u>ffff</u>	<u>ffff</u>	<u>ffff</u>
<u>Description:</u>	<u>Move the contents of the working register into the file register.</u> <u>The 'B' bit selects byte or word operation.</u> <u>The 'f' bits select the address of the file register.</u> <u>Note: The extension .b in the instruction denotes a byte move rather than a word move. You may use a .w extension to denote a word move, but it is not required.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1</u>					
<u>Example:</u>		<u>MOV</u>	<u>Ww,213</u>	<u>: Move Ww to File Register 213</u>		
	<u>Before Instruction</u> <u>After Instruction</u>					

TABLE 96 -- MPY: Multiply Wm by Wn to Accumulator

Syntax:	{label:}MPY	A, B,	Wm*Wn	,Wxp,[Wx] ,Wxp,[Wx]+=kx ,Wxp,[Wx]-=kx ‡ ,Wxp,[W5+W8] none	,Wyp,[Wy] ,Wyp,[Wy]+=ky ,Wyp,[Wy]-=ky ‡ ,Wyp,[W7+W8] none	
				‡ Alternate format for negative kx,ky		
Operands:	Wm*Wn ∈ {W0*W1; W0*W2; W0*W3; W1*W2; W1*W3; W2*W3} Wxp ∈ {W0 ... W3}; Wx ∈ {W4, W5}; kx ∈ {-6, -4, -2, 2, 4, 6}; Wyp ∈ {W0 ... W3}; Wy ∈ {W6, W7}; ky ∈ {-6, -4, -2, 2, 4, 6}; AWB ∈ {W9, [W9]++}					
Operation:	(Wm)*(Wn) → ACC(A or B); ([Wx]) → Wxp; (Wx)+kx → Wx; ([Wy]) → Wyp; (Wy)+ky → Wy;					
Status Affected:	OA, OB, SA, SB					
Encoding:	1100	Omnm	A0xx	yyii	ijjj	jj11
Description:	<u>Signed, fractional or integer multiply the contents of two W registers. The 32-bit result is sign-extended to 40-bits and stored to the specified accumulator.</u> <u>Wx register specifies the prefetch of the multiplier Wxp register. The prefetch is done with indirect, indirect with post inc/dec, indirect with register offset, copy of the other prefetch or none. Post-modify Wx as required.</u> <u>Wy register specifies the prefetch of the multiplier Wyp register. Post-modify Wy as required.</u> <u>The ‘m’ bits select the operand registers Wm and Wn for the multiply;</u> <u>The ‘A’ bit selects the accumulator for the result.</u> <u>The ‘i’ bits select the Wx pre-fetch operation.</u> <u>The ‘j’ bits select the Wy pre-fetch operation.</u> <u>The ‘x’ bits select the pre-fetch Wxp destination.</u> <u>The ‘y’ bits select the pre-fetch Wyp destination.</u> <u>See Table 1-9 through Table 1-13 for modifier addressing information.</u>					
Words:	1					
Cycles:	1					
Example:	MPY	A,W2*W3,W0,[W5]-=6,W1,[W7]			; Multiply into Accumulator A	
	Before Instruction					
	ACCA = 2					
	ACCB = 3					
	W0 = 5					
	W1 = 6					

	<u>W2 = 7</u> <u>W3 = 8</u> <u>W8 = 1000</u> <u>W10 = 2000</u> <u>RAM(994) = 16</u> <u>RAM(1000) = 17</u> <u>RAM(2000) = 18</u> <u>After Instruction</u> <u>ACCA = 7*8=56</u> <u>ACCB = 3</u> <u>W0 = 17</u> <u>W1 = 18</u> <u>W2 = 7</u> <u>W3 = 8</u> <u>W8 = 994</u> <u>W10 = 2000</u> <u>RAM(994) = 16</u> <u>RAM(1000) = 17</u> <u>RAM(2000) = 18</u>	
--	--	--

TABLE 97 -- MPYN: Multiply -Wm by Wn to Accumulator

Syntax:	{label:}MPYN	A, B,	Wm*Wn	,Wxp,[Wx] ,Wxp,[Wx]+=kx ,Wxp,[Wx]-=kx ‡ ,Wxp,[W5+W8] none	,Wyp,[Wy] ,Wyp,[Wy]+=ky ,Wyp,[Wy]-=ky ‡ ,Wyp,[W7+W8] none	
				‡ Alternate format for negative kx,ky		
Operands:	Wm*Wn ∈ {W0*W1; W0*W2; W0*W3; W1*W2; W1*W3; W2*W3} Wxp ∈ {W0 ... W3}; Wx ∈ {W4, W5}; kx ∈ {-6, -4, -2, 2, 4, 6}; Wyp ∈ {W0 ... W3}; Wy ∈ {W6, W7}; ky ∈ {-6, -4, -2, 2, 4, 6}; AWB ∈ {W9, [W9]++}					
Operation:	-(Wm)*(Wn) → ACC(A or B); ([Wx]) → Wxp; (Wx)+kx → Wx; ([Wy]) → Wyp; (Wy)+ky → Wy;					
Status	OA, OB, SA, SB					
Affected:						
Encoding:	1100	0mmmm	A1xx	yyii	ijjj	jj11
Description:	<p>Signed, fractional or integer multiply the contents of a W register by the negative of the contents of another W register. The 32-bit result is sign-extended to 40-bits and stored to the specified accumulator.</p> <p>Wx register specifies the prefetch of the multiplier Wxp register. The prefetch is done with indirect, indirect with post inc/dec, indirect with register offset, copy of the other prefetch or none. Post-modify Wx as required.</p> <p>Wy register specifies the prefetch of the multiplier Wyp register. Post-modify Wy as required.</p> <p>The 'm' bits select the operand registers Wm and Wn for the multiply;</p> <p>The 'A' bit selects the accumulator for the result.</p> <p>The 'i' bits select the Wx pre-fetch operation.</p> <p>The 'j' bits select the Wy pre-fetch operation.</p> <p>The 'x' bits select the pre-fetch Wxp destination.</p> <p>The 'y' bits select the pre-fetch Wyp destination.</p> <p>See Table 1-9 through Table 1-13 for modifier addressing information.</p>					
Words:	1					
Cycles:	1					
Example:	MPYN	A,W2*W3,W0,[W4]-=6,W1,[W6]	; Multiply negative into Acc A			
	Before Instruction					
	ACCA = 2					
	ACCB = 3					
	W0 = 5					
	W1 = 6					
	W2 = 7					
	W3 = 8					
	W8 = 1000					

	<u>W10 = 2000</u> <u>RAM(994) = 16</u> <u>RAM(1000) = 17</u> <u>RAM(2000) = 18</u> <u>After Instruction</u> <u>ACCA = -7*8=-56</u> <u>ACCB = 3</u> <u>W0 = 17</u> <u>W1 = 18</u> <u>W2 = 7</u> <u>W3 = 8</u> <u>W8 = 994</u> <u>W10 = 2000</u> <u>RAM(994) = 16</u> <u>RAM(1000) = 17</u> <u>RAM(2000) = 18</u>	
--	--	--

TABLE 98 -- MSC: Multiply and Subtract from Accumulator

<u>Syntax:</u>	<u>{label;}MSC</u>	<u>A,</u>	<u>Wm*Wn</u>	<u>,Wxp,[Wx]</u>	<u>,Wyp,[Wy]</u>	<u>,AWB</u>
		<u>B,</u>		<u>,Wxp,[Wx]+=kx</u>	<u>,Wyp,[Wy]+=ky</u>	<u>none</u>
				<u>,Wxp,[Wx]-=kx ‡</u>	<u>,Wyp,[Wy]-=ky ‡</u>	
				<u>,Wxp,[W5+W8]</u>	<u>,Wyp,[W7+W8]</u>	
				<u>none</u>	<u>none</u>	
				<u>‡ Alternate format for negative kx,ky</u>		
<u>Operands:</u>	<u>Wm*Wn ∈ {W0*W1; W0*W2; W0*W3; W1*W2; W1*W3; W2*W3}</u> <u>Wxp ∈ {W0 ... W3}; Wx ∈ {W4, W5}; kx ∈ {-6, -4, -2, 2, 4, 6};</u> <u>Wyp ∈ {W0 ... W3}; Wy ∈ {W6, W7}; ky ∈ {-6, -4, -2, 2, 4, 6};</u> <u>AWB ∈ {W9, [W9]++}</u>					
<u>Operation:</u>	<u>(ACC(A or B)) - (Wm)*(Wn) → ACC(A or B);</u> <u>([Wx]) → Wxp; (Wx)+kx → Wx;</u> <u>([Wy]) → Wyp; (Wy)+ky → Wy;</u> <u>(ACC(B or A)) rounded → AWB</u>					
<u>Status Affected:</u>	<u>OA, OB, SA, SB</u>					
<u>Encoding:</u>	<u>1100</u>	<u>0mmmm</u>	<u>A1xxx</u>	<u>yyii</u>	<u>iijj</u>	<u>jjaaa</u>
<u>Description:</u>	<u>Signed, fractional or integer multiply the contents of two W registers. The 32-bit result is sign-extended to 40-bits and subtracted from the specified accumulator.</u> <u>Wx register specifies the prefetch of the multiplier Wxp register. The prefetch is done with indirect, indirect with post inc/dec, indirect with register offset, copy of the other prefetch or none. Post-modify Wx as required.</u> <u>Wy register specifies the prefetch of the multiplier Wyp register. Post-modify Wy as required.</u> <u>AWB specifies the direct or indirect store of the convergently rounded contents of other accumulator, if required.</u> <u>The 'm' bits select the operand registers Wm and Wn for the multiply;</u> <u>The 'A' bit selects the accumulator for the result. The other accumulator is used for write back.</u> <u>The 'i' bits select the Wx pre-fetch operation.</u> <u>The 'j' bits select the Wy pre-fetch operation.</u> <u>The 'x' bits select the pre-fetch Wxp destination.</u> <u>The 'y' bits select the pre-fetch Wyp destination.</u> <u>The 'a' bits select the accumulator write-back destination.</u> <u>See Table 1-9 through Table 1-14 for modifier addressing information.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1</u>					
<u>Example:</u>	<u>MSC</u>	<u>A,W2*W3,W0=[W4]-=6,W1=[W6],W9</u>				<u>; Multiply and Subtract A</u>

	<u>Before Instruction</u> <u>ACCA = 2</u> <u>ACCB = 3</u> <u>W0 = 5</u> <u>W1 = 6</u> <u>W2 = 7</u> <u>W3 = 8</u> <u>W8 = 1000</u> <u>W10 = 2000</u> <u>RAM(994) = 16</u> <u>RAM(1000) = 17</u> <u>RAM(2000) = 18</u> <u>After Instruction</u> <u>ACCA = 2+7*8=58</u> <u>ACCB = 3</u> <u>W0 = 17</u> <u>W1 = 18</u> <u>W2 = 7</u> <u>W3 = 8</u> <u>W8 = 994</u> <u>W10 = 2000</u> <u>RAM(994) = 3</u> <u>RAM(1000) = 17</u> <u>RAM(2000) = 18</u>	
--	--	--

TABLE 99 -- MULS: 16x16 bit Signed Multiply

<u>Syntax:</u>	<u>{label:}</u>	<u>MUL.SS</u>	<u>Wb,</u>	<u>Ws,</u>	<u>Wnd</u>	
				<u>[Ws],</u>		
				<u>[Ws]++,</u>		
				<u>[Ws]--,</u>		
				<u>[Ws++]</u> ,		
				<u>[Ws--]</u> ,		
<u>Operands:</u>	<u>Wb ∈ [W0 ... W15];</u> <u>Ws ∈ [W0 ... W15];</u> <u>Wnd ∈ [W0,W2,W4,W6,W8,W10,W12,W14]</u>					
<u>Operation:</u>	<u>signed (Wb) * signed (Ws) → {Wnd+1, Wnd}</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>1011</u>	<u>1001</u>	<u>1www</u>	<u>wddd</u>	<u>dppp</u>	<u>ssss</u>
<u>Description:</u>	<u>MULS performs a 16-bit x 16-bit multiply, with the result stored in two successive working registers.</u> <u>Both source operands are interpreted as two's-complement signed integers.</u> <u>The 'w' bits select the address of the base register</u> <u>The 's' bits select the address of the source register.</u> <u>The 'p' bits select source address mode 2.</u> <u>The 'd' bits select the address of the destination for the product LSBs,</u> <u>the register 'd+1' is the destination of the product MSBs.</u> <u>See Table 1-5 for modifier addressing information.</u> <u>Note: This instruction operates in word mode only.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1</u>					
<u>Example:</u>		<u>MUL.SS</u>	<u>W5, W6, W8</u>	<u>; Multiply W5*W6 to W9:W8</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 100 -- MULSU: 16x16 bit Signed-Unsigned Multiply

<u>Syntax:</u>	<u>{label:}</u>	<u>MUL.SU</u>	<u>Wb,</u>	<u>Ws,</u>	<u>Wnd</u>	
				<u>[Ws],</u>		
				<u>[Ws]++,</u>		
				<u>[Ws]--,</u>		
				<u>[Ws++]</u> ,		
				<u>[Ws--]</u> ,		
<u>Operands:</u>	<u>Wb</u> \in <u>[W0 ... W15];</u> <u>Ws</u> \in <u>[W0 ... W15];</u> <u>Wnd</u> \in <u>[W0,W2,W4,W6,W8,W10,W12,W14]</u>					
<u>Operation:</u>	<u>signed (Wb) * unsigned (Ws) \rightarrow {Wnd+1, Wnd}</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>1011</u>	<u>1001</u>	<u>0www</u>	<u>wddd</u>	<u>dppp</u>	<u>ssss</u>
<u>Description:</u>	<u>MULSU performs a 16-bit x 16-bit multiply, with the result stored in two successive working registers.</u> <u>The first source operands is interpreted as a two's-complement signed integer and the second source operand is interpreted as an unsigned integer.</u> <u>The 'w' bits select the address of the base register</u> <u>The 's' bits select the address of the source register.</u> <u>The 'p' bits select source address mode 2.</u> <u>The 'd' bits select the address of the destination for the product LSBs, the register 'd+1' is the destination of the product MSBs.</u> <u>See Table 1-5 for modifier addressing information.</u> <u>Note: This instruction operates in word mode only.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1</u>					
<u>Example:</u>		<u>MUL.SU</u>	<u>W5, W6, W8</u>	<u>: Multiply W5*W6 to W9:W8</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 101 -- MULSULS: 16x16 bit Signed Multiply Unsigned Short Literal

Syntax:	{label:}	MUL.SU	Wb,	lit5,	Wnd	
Operands:	Wb ∈ [W0 ... W15]; lit5 ∈ [0 ... 31]; Wnd ∈ [W0,W2,W4,W6,W8,W10,W12,W14]					
Operation:	signed (Wb) * unsigned lit5 → {Wnd+1, Wnd}					
Status Affected:	None					
Encoding:	1011	1001	1www	wddd	d11k	kkkk
Description:	<p>MULSLS performs a 16-bit x 16-bit multiply, with the result stored in two successive working registers.</p> <p>The source operands is interpreted as a two's-complement signed integer and the literal is interpreted as an unsigned integer.</p> <p>The 'k' bits define a 5-bit unsigned integer literal.</p> <p>The 'w' bits select the address of the base register.</p> <p>The 'd' bits select the address of the destination for the product LSBs, the register 'd+1' is the destination of the product MSBs.</p> <p>Note: This instruction operates in word mode only.</p>					
Words:	1					
Cycles:	1					
Example:		MUL.SU	W6, #13, W8	: Multiply W6 times 13 into W9:W8		
	Before Instruction					
	After Instruction					

TABLE 102 -- MULU: 16x16 bit Unsigned Multiply

<u>Syntax:</u>	<u>{label:}</u>	<u>MUL.UU</u>	<u>Wb,</u>	<u>Ws,</u>	<u>Wnd</u>	
				<u>[Ws],</u>		
				<u>[Ws]++,</u>		
				<u>[Ws]--,</u>		
				<u>[Ws++] ,</u>		
				<u>[Ws--] ,</u>		
<u>Operands:</u>	<u>Wb ∈ [W0 ... W15]:</u> <u>Ws ∈ [W0 ... W15]:</u> <u>Wnd ∈ [W0,W2,W4,W6,W8,W10,W12,W14]</u>					
<u>Operation:</u>	<u>unsigned (Wb) * unsigned (Ws) → {Wnd+1, Wnd}</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>1011</u>	<u>1000</u>	<u>0www</u>	<u>wddd</u>	<u>dppp</u>	<u>ssss</u>
<u>Description:</u>	<u>MULU performs a 16-bit x 16-bit multiply, with the result stored in two successive working registers.</u> <u>Both source operands are interpreted as unsigned integers.</u> <u>The ‘w’ bits select the address of the base register.</u> <u>The ‘s’ bits select the address of the source register.</u> <u>The ‘p’ bits select source address mode 2.</u> <u>The ‘d’ bits select the address of the destination for the product LSBs, the register ‘d+1’ is the destination of the product MSBs.</u> <u>See Table 1-5 for modifier addressing information.</u> <u>Note: This instruction operates in word mode only.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1</u>					
<u>Example:</u>		<u>MUL.UU</u>	<u>W5, W6, W8</u>	<u>: Multiply W5*W6 to W9:W8</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 103 -- MULULS: 16x16 bit Unsigned Multiply Short Literal

Syntax:	{label:}	MULULS	Wb,	lit5,	Wnd	
Operands:	Wb ∈ [W0 ... W15]; lit5 ∈ [0 ... 31]; Wnd ∈ [W0,W2,W4,W6,W8,W10,W12,W14]					
Operation:	unsigned (Wb) * unsigned lit5 → {Wnd+1, Wnd}					
Status Affected:	None					
Encoding:	1011	1000	0www	wddd	d11k	kkkk
Description:	<p>MULULS performs a 16-bit x 16-bit multiply, with the result stored in two successive working registers.</p> <p>Both operands are interpreted as unsigned integers.</p> <p>The 'k' bits define a 5-bit unsigned integer literal.</p> <p>The 'w' bits select the address of the base register.</p> <p>The 'd' bits select the address of the destination for the product LSBs, the register 'd+1' is the destination of the product MSBs.</p> <p>Note: This instruction operates in word mode only.</p>					
Words:	1					
Cycles:	1					
Example:	MUL.UU W6, #13, W8 : Multiply W6 times 13 into W9:W8					
	Before Instruction After Instruction					

TABLE 104 -- MULUS: 16x16 bit Unsigned-Signed Multiply

<u>Syntax:</u>	<u>{label:}</u>	<u>MUL.US</u>	<u>Wb,</u>	<u>Ws,</u>	<u>Wnd</u>	
				<u>[Ws],</u>		
				<u>[Ws]++,</u>		
				<u>[Ws]--,</u>		
				<u>[Ws++]</u> ,		
				<u>[Ws--],</u>		
<u>Operands:</u>	<u>Wb ∈ [W0 ... W15];</u> <u>Ws ∈ [W0 ... W15];</u> <u>Wnd ∈ [W0,W2,W4,W6,W8,W10,W12,W14]</u>					
<u>Operation:</u>	<u>unsigned (Wb) * signed (Ws) → {Wnd+1, Wnd}</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>1011</u>	<u>1000</u>	<u>1www</u>	<u>wddd</u>	<u>dppp</u>	<u>ssss</u>
<u>Description:</u>	<u>MULUS performs a 16-bit x 16-bit multiply, with the result stored in two successive working registers.</u> <u>The first source operands is interpreted as an unsigned integer and the second source operand is interpreted as a two's-complement signed integer.</u> <u>The 'w' bits select the address of the base register.</u> <u>The 's' bits select the address of the source register.</u> <u>The 'p' bits select source address mode 2.</u> <u>The 'd' bits select the address of the destination for the product LSBs, the register 'd+1' is the destination of the product MSBs.</u> <u>See Table 1-5 for modifier addressing information.</u> <u>Note: This instruction operates in word mode only.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1</u>					
<u>Example:</u>		<u>MUL.US</u>	<u>W5, W6, W8</u>	<u>: Multiply W5*W6 to W9:W8</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 105 -- MULWF: 8-bit x 8-bit Multiply

Syntax:	<u>{label:}</u>	<u>MUL{.b}</u>	<u>f</u>			
Operands:	<u>f ∈ [0 ... 8191]</u>					
Operation:	<u>If byte mode, (Ww)<7:0> * (f)<7:0> → W2</u> <u>If word mode, (Ww) * (f) → W3:W2</u>					
Status Affected:	<u>None</u>					
Encoding:	<u>1011</u>	<u>1100</u>	<u>0B0f</u>	<u>ffff</u>	<u>ffff</u>	<u>ffff</u>
Description:	<u>Multiply the working register and the file register and place the result in the W3:W2 register pair.</u> <u>The 'B' bit selects byte or word operation.</u> <u>The 'f' bits select the address of the file register.</u> <u>Note: Word operation is assumed.</u>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>MUL</u>	<u>RAM135</u>	<u>; Multiply Ww by RAM135</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 106 -- NEG: Negate Ws

Syntax:	{label:}	NEG{.b}	Ws,	Wd		
			<u>[Ws],</u>	<u>[Wd]</u>		
			<u>[Ws]++</u> ,	<u>[Wd]++</u>		
			<u>[Ws]--</u> ,	<u>[Wd]--</u>		
			<u>[Ws++]</u> ,	<u>[Wd++]</u>		
			<u>[Ws--]</u> ,	<u>[Wd--]</u>		
Operands:	<u>Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]</u>					
Operation:	<u>(Ws)+ 1 → Wd</u>					
Status Affected:	<u>C, DC, N, OV, Z</u>					
Encoding:	<u>1110</u>	<u>1010</u>	<u>0Bqq</u>	<u>qddd</u>	<u>dppp</u>	<u>ssss</u>
Description:	<p><u>Compute the 2's complement of the contents of the source register Ws and place the result in the destination register Wd.</u></p> <p><u>The 'B' bit selects byte or word operation.</u></p> <p><u>The 's' bits select the address of the source register.</u></p> <p><u>The 'd' bits select the address of the destination register.</u></p> <p><u>The 'p' bits select the source address mode 2 (values 0-4).</u></p> <p><u>The 'q' bits select the destination address mode 2 (values 0-4).</u></p> <p><u>See Table 1-5 and Table 1-6 for modifier addressing information.</u></p> <p><u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:	<u>NEG</u>	<u>W5,W7</u>	<u>; Negate</u>			
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 107 -- NEGAB: Negate Accumulators

Syntax:	<u>{label:}</u>	<u>NEG</u>	<u>A</u>			
			<u>B</u>			
Operands:	<u>None</u>					
Operation:	<u>if (NEGAB A) then -ACCA → ACCA</u> <u>if (NEGAB B) then -ACCB → ACCB</u>					
Status Affected:	<u>OA, OB, SA, SB</u>					
Encoding:	<u>1100</u>	<u>1011</u>	<u>A001</u>	<u>0000</u>	<u>0000</u>	<u>0000</u>
Description:	<u>Negate Accumulator.</u> <u>The 'A' bits specify the selected accumulator.</u>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>NEG</u>	<u>B</u>	<u>; Negate ACCB, result to ACCB</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 108 -- NEGF: Negate f

Syntax:	<u>{label:}</u>	<u>NEG{.b}</u>	<u>f</u>	<u>{,Ww}</u>		
Operands:	<u>f ∈ [0 ... 8191]</u>					
Operation:	<u>(f) + 1 → destination designated by D</u>					
Status Affected:	<u>C, DC, N, OV, Z</u>					
Encoding:	<u>1110</u>	<u>1110</u>	<u>0Bdf</u>	<u>ffff</u>	<u>ffff</u>	<u>ffff</u>
Description:	<p><u>Compute the 2's complement of the contents of the file register and place the result in the destination designated by D. If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.</u></p> <p><u>The 'B' bit selects byte or word operation.</u></p> <p><u>The 'f' bits select the address of the file register.</u></p> <p><u>The 'D' bit selects the destination.</u></p> <p><u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>NEG</u>		<u>RAM135</u>		<u>; Negate</u>
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 109 -- NOP: No Operation

<u>Syntax:</u>	<u>{label:}</u>	<u>NOP</u>				
<u>Operands:</u>	<u>None</u>					
<u>Operation:</u>	<u>No Operation</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>0000</u>	<u>0000</u>	<u>xxxx</u>	<u>xxxx</u>	<u>xxxx</u>	<u>xxxx</u>
<u>Description:</u>	<u>No Operation is performed.</u>					
	<u>The 'x' bits can take any value.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1</u>					
<u>Example:</u>		<u>NOP</u>			<u>; No operation</u>	
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 110 -- NOPR: No Operation

Syntax:	{label:}	NOPR				
Operands:	None					
Operation:	No Operation					
Status Affected:	None					
Encoding:	1111	1111	xxxx	xxxx	xxxx	xxxx
Description:	No Operation is performed.					
	The 'x' bits can take any value.					
Words:	1					
Cycles:	1					
Example:		NOPR			No Opeation	
	Before Instruction					
	After Instruction					

TABLE 111 -- POP: Pop top of Return Stack

Syntax:	<u>{label:}</u>	<u>POP</u>	<u>f</u>			
Operands:	<u>$f \in [0 \dots 65534]$</u>					
Operation:	<u>$(W15)+2 \rightarrow W15$</u> <u>$(TOS) \rightarrow f$</u>					
Status Affected:	<u>None</u>					
Encoding:	<u>1111</u>	<u>1001</u>	<u>ffff</u>	<u>ffff</u>	<u>ffff</u>	<u>ffff</u>
Description:	<u>The stack pointer (W15) is pre-incremented and Top of Stack (TOS) value is pulled off the stack and written to the file register.</u> <u>Note: This instruction operates in word mode only.</u>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>POP</u>		<u>RAM135</u>		<u>: Pop</u>
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 112 -- PUSH: Push top of return stack (TOS)

Syntax:	<u>{label:}</u>	<u>PUSH</u>	<u>f</u>			
Operands:	<u>f ∈ [0 ... 65534]</u>					
Operation:	<u>f → TOS</u> <u>(W15)-2 → W15</u>					
Status Affected:	<u>None</u>					
Encoding:	<u>1111</u>	<u>1000</u>	<u>ffff</u>	<u>ffff</u>	<u>ffff</u>	<u>ffff</u>
Description:	<u>The file register contents are written to the Top of Stack (TOS) location. Then the stack pointer (W15) is post decremented.</u> <u>Note: This instruction operates in word mode only.</u>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>PUSH</u>	<u>RAM135</u>	<u>:</u>	<u>Push</u>	
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 113 -- RCALL: Relative Call

<u>Syntax:</u>	<u>{label:}</u>	<u>RCALL</u>	<u>Slit16</u>			
<u>Operands:</u>	<u>Slit16 ∈ [-32768 ... +32767]</u>					
<u>Operation:</u>	<u>(PC) +2→PC,</u> <u>(PC<15:0>)→TOS,</u> <u>(W15)+2 → W15</u> <u>(PC<23:16>)→TOS,</u> <u>(W15)+2 → W15</u> <u>(PC)+(2 * Slit16)→PC, NOP→Instruction Register.</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>0000</u>	<u>0111</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>	<u>nnnn</u>
<u>Description:</u>	<u>Subroutine call with a jump up to 32K instructions from the current location.</u> <u>First, return address (PC+2) is pushed onto the return stack (20-bits wide).</u> <u>Then the sign extended 17-bit value (2 * Slit16) is added to the contents of the PC and the result is stored into the PC. RCALL is a two-cycle instruction.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>2</u>					
<u>Example:</u>		<u>RCALL</u>	<u>label</u>	:	<u>Call subroutine</u>	
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 114 -- RCALLW: Computed Call

<u>Syntax:</u>	<u>{label:}</u>	<u>RCALL</u>	<u>Wn</u>			
<u>Operands:</u>	<u>Wn ∈ [W0 ... W15]</u>					
<u>Operation:</u>	<u>(PC) + 2 → PC,</u> <u>(PC<15:0>) → TOS,</u> <u>(W15)+2 → W15</u> <u>(PC<23:16>) → TOS,</u> <u>(W15)+2 → W15</u> <u>(PC) + (2 * (Wn)) → PC, NOP → Instruction Register.</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>0000</u>	<u>0001</u>	<u>0010</u>	<u>0000</u>	<u>0000</u>	<u>ssss</u>
<u>Description:</u>	<u>Computed subroutine call with a jump up to 32K instructions forward or back from the current location. First, return address (PC+2) is pushed onto the return stack.</u> <u>Then the sign extended 17-bit value (2 * (Wn)) is added to the contents of the PC and the result is stored into the PC. RCALLW is a two-cycle instruction.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>2</u>					
<u>Example:</u>		<u>RCALL</u>	<u>W11</u>	<u>: Call subroutine at PC+W11</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 115 -- REPEAT: Repeat next instruction n times

Syntax:	<u>{label:}</u>	<u>REPEAT</u>	<u>lit14</u>			
Operands:	<u>lit14</u> ∈ [1 ... 16383]					
Operation:	<u>(lit14) → LCR (Loop Count Register)</u> <u>(PC)+2 → PC</u> <u>Enable Code Looping</u>					
Status Affected:	<u>None</u>					
Encoding:	<u>0000</u>	<u>1001</u>	<u>00kk</u>	<u>kkkk</u>	<u>kkkk</u>	<u>kkkk</u>
Description:	<p><u>The instruction immediately following the REPEAT instruction is repeated lit14 times. The repeated instruction is held in the instruction register for all iterations and so is fetched only once (during the REPEAT instruction, as would be expected). The first iteration of the repeated instruction pre-fetches the next instruction.</u></p> <p><u>The repeat count is decremented during each iteration. When it equals zero, the pre-fetch instruction is staged into the instruction and normal execution continues.</u></p> <p><u>The repeated instruction can be interrupted before any iteration, but only by a priority 1 (fast context switch) interrupt. Subsequent interrupts must be held pending until the repeat operation is complete. Note that nested repeats (e.g. from within the interrupt service routine) are not supported.</u></p> <p><u>The 'k' bits are an unsigned literal that specifies the loop count.</u></p>					
Words:	<u>1</u>					
Cycles:	<u>1 + lit14</u>					
Example:		<u>REPEAT</u>	<u>#5</u>	<u>; Repeat next instruction 5 times</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 116 -- REPEATW: Repeat next instruction Wn times

Syntax:	{label:}	REPEAT	Wn			
Operands:	Wn ∈ [W0 ... W15]					
Operation:	(Wn) → LCR (Loop Count Register) (PC)+2 → PC Enable Code Looping					
Status Affected:	None					
Encoding:	0000	1001	1000	0000	0000	ssss
Description:	<p><u>The instruction immediately following the REPEAT instruction is repeated (Wn) times. The repeated instruction is held in the instruction register for all iterations and so is fetched only once (during the REPEAT instruction, as would be expected). The first iteration of the repeated instruction pre-fetches the next instruction.</u></p> <p><u>The repeat count is decremented during each iteration. When it equals zero, the pre-fetch instruction is staged into the instruction and normal execution continues.</u></p> <p><u>The repeated instruction can be interrupted before any iteration, but only by a priority 1 (fast context switch) interrupt. Subsequent interrupts must be held pending until the repeat operation is complete. Note that nested repeats (e.g. from within the interrupt service routine) are not supported.</u></p> <p><u>The 's' bits specify the Wn register that contains the loop count.</u></p>					
Words:	1					
Cycles:	1 + (Wn)					
Example:		REPEAT	W6	; Repeat next instruction (W6) times		
	Before Instruction					
	After Instruction					

TABLE 117 -- RESET: Reset

Syntax:	{label:}	RESET				
Operands:	none					
Operation:	Force all registers and flag bits that are affected by M68LR reset to their reset condition.					
Status Affected:	None					
Encoding:	1111	1110	0000	0000	0000	0000
Description:	This instruction provides a way to execute a software reset.					
Words:	1					
Cycles:	1					
Example:		RESET			: Reset	
	Before Instruction					
	After Instruction					

TABLE 118 -- RLC: Rotate Left Ws through Carry

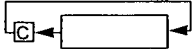
Syntax:	{label:}	RLC{.b}	Ws,	Wd		
			[Ws],	[Wd]		
			[Ws]++,	[Wd]++		
			[Ws]--,	[Wd]--		
			[Ws++] ,	[Wd++]		
			[Ws--] ,	[Wd--]		
Operands:	Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]					
Operation:	<p>For word operation: (C) → Wd<0>, (Ws<14:0>) → Wd<15:1>, (Ws<15>) → C</p> <p>For byte operation: (C) → Wd<0>, (Ws<6:0>) → Wd<7:1>, (Ws<7>) → C</p> 					
Status Affected:	C, N, Z					
Encoding:	1101	0010	1Bqq	qddd	dppp	ssss
Description:	<p>Rotate the contents of the source register Ws one bit to the left through the carry flag and place the result in the destination register Wd.</p> <p>The 'B' bit selects byte or word operation.</p> <p>The 's' bits select the address of the source register.</p> <p>The 'd' bits select the address of the destination register.</p> <p>The 'p' bits select source address mode 2.</p> <p>The 'q' bits select destination address mode 2.</p> <p>See Table 1-5 and Table 1-6 for modifier addressing information.</p> <p>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</p>					
Words:	1					
Cycles:	1					
Example:		RLC	W5,W6	; Rotate left		
	Before Instruction					
	After Instruction					

TABLE 119 -- RLCF: Rotate Left f through Carry

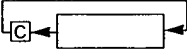
Syntax:	{label:}	RLC{b}	f	{,Ww}		
Operands:	f ∈ [0 ... 8191]					
Operation:	For word operation: (C) → Dest<0>, (f<14:0>) → Dest<15:1>, (f<15>) → C For byte operation: (C) → Dest<0>, (f<6:0>) → Dest<7:1>, (f<7>) → C 					
Status Affected:	C, N, Z					
Encoding:	1101	0110	1BDf	ffff	ffff	ffff
Description:	<p>Rotate the contents of the file register f one bit to the left through the carry flag and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.</p> <p>The 'B' bit selects byte or word operation. The 'D' bit selects the destination. The 'f' bits select the address of the file register.</p> <p>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</p>					
Words:	1					
Cycles:	1					
Example:		RLC	RAM135, Ww	; Rotate left		
	Before Instruction					
	After Instruction					

TABLE 120 -- RLNC: Rotate Left Ws (No Carry)

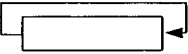
Syntax:	<u>{label:}</u>	<u>RLNC{.b}</u>	<u>Ws,</u>	<u>Wd</u>		
			<u>[Ws],</u>	<u>[Wd]</u>		
			<u>[Ws]++,</u>	<u>[Wd]++</u>		
			<u>[Ws]--,</u>	<u>[Wd]--</u>		
			<u>[Ws++]</u> ,	<u>[Wd++]</u>		
			<u>[Ws--]</u> ,	<u>[Wd--]</u>		
Operands:	<u>Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]</u>					
Operation:	<u>For word operation:</u> <u>(Ws<14:0>) → Wd<15:1>, (Ws<15>) → Wd<0></u> <u>For byte operation:</u> <u>(Ws<6:0>) → Wd<7:1>, (Ws<7>) → Wd<0></u> 					
Status Affected:	<u>N, Z</u>					
Encoding:	<u>1101</u>	<u>0010</u>	<u>0Bqq</u>	<u>qddd</u>	<u>dppp</u>	<u>ssss</u>
Description:	<u>Rotate the contents of the source register Ws one bit to the left and place the result in the destination register Wd. The Carry Flag bit is not affected.</u> <u>The 'B' bit selects byte or word operation.</u> <u>The 's' bits select the address of the source register.</u> <u>The 'd' bits select the address of the destination register.</u> <u>The 'p' bits select source address mode 2.</u> <u>The 'q' bits select destination address mode 2.</u> <u>See Table 1-5 and Table 1-6 for modifier addressing information.</u> <u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>RLNC</u>	<u>W5,W6</u>	<u>: Rotate left</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 121 -- RLNCF: Rotate Left f (No Carry)

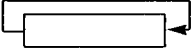
Syntax:	<u>{label:}</u>	<u>RLNC{.b}</u>	<u>f</u>	<u>{,Ww}</u>		
Operands:	<u>f ∈ [0 ... 8191]</u>					
Operation:	<u>For word operation:</u> <u>(f<14:0>) → Dest<15:1>, (f<15>) → Dest<0></u> <u>For byte operation:</u> <u>(f<6:0>) → Dest<7:1>, (f<7>) → Dest<0></u> 					
Status Affected:	<u>N, Z</u>					
Encoding:	<u>1101</u>	<u>0110</u>	<u>0BDf</u>	<u>ffff</u>	<u>ffff</u>	<u>ffff</u>
Description:	<u>Rotate the contents of the file register f one bit to the left and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register. The carry flag bit is not affected.</u> <u>The 'B' bit selects byte or word operation.</u> <u>The 'D' bit selects the destination.</u> <u>The 'f' bits select the address of the file register.</u> <u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>RLNC</u>	<u>RAM135, Ww</u>	<u>; Rotate left</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 122 -- RRC: Rotate Right Ws through Carry

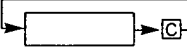
Syntax:	{label:}	RRC{.b}	Ws,	Wd		
			<u>[Ws],</u>	<u>[Wd]</u>		
			<u>[Ws]++,</u>	<u>[Wd]++</u>		
			<u>[Ws]--,</u>	<u>[Wd]--</u>		
			<u>[Ws++]</u> ,	<u>[Wd++]</u>		
			<u>[Ws--]</u> ,	<u>[Wd--]</u>		
Operands:	Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]					
Operation:	For word operation: (C) → Wd<15>, (Ws<15:1>) → Wd<14:0>, (Ws<0>) → C For byte operation: (C) → Wd<7>, (Ws<7:1>) → Wd<6:0>, (Ws<0>) → C 					
Status Affected:	C, N, Z					
Encoding:	<u>1101</u>	<u>0011</u>	<u>1Bqq</u>	<u>qddd</u>	<u>dppp</u>	<u>ssss</u>
Description:	<u>Rotate the contents of the source register Ws one bit to the right through the carry flag and place the result in the destination register Wd.</u> <u>The 'B' bit selects byte or word operation.</u> <u>The 's' bits select the address of the source register.</u> <u>The 'd' bits select the address of the destination register.</u> <u>The 'p' bits select source address mode 2.</u> <u>The 'q' bits select destination address mode 2.</u> <u>See Table 1-5 and Table 1-6 for modifier addressing information.</u> <u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>RRC</u>	<u>W5,W6</u>	<u>: Rotate right</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 123 -- RRCF: Rotate Right f through Carry

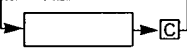
Syntax:	{label:}	RRC{.b}	f	{,Ww}		
Operands:	f ∈ [0 ... 8191]					
Operation:	For word operation: (C) → Dest<15>, (f<15:1>) → Dest<14:0>, (f<0>) → C For byte operation: (C) → Dest<7>, (f<7:1>) → Dest<6:0>, (f<0>) → C 					
Status Affected:	C, N, Z					
Encoding:	1101	0111	1B Df	ffff	ffff	ffff
Description:	<p>Rotate the contents of the file register f one bit to the left through the carry flag and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register..</p> <p>The 'B' bit selects byte or word operation. The 'D' bit selects the destination. The 'f' bits select the address of the file register.</p> <p>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</p>					
Words:	1					
Cycles:	1					
Example:		RRC	RAM135, Ww	; Rotate right		
	Before Instruction					
	After Instruction					

TABLE 124 -- RRNC: Rotate Right Ws (No Carry)

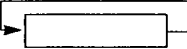
Syntax:	{label:}	RRNC{.b}	Ws,	Wd		
			[Ws],	[Wd]		
			[Ws]++,	[Wd]++		
			[Ws]--,	[Wd]--		
			[Ws++] ,	[Wd++]		
			[Ws--] ,	[Wd--]		
Operands:	Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]					
Operation:	For word operation: (Ws<15:1>) → Wd<14:0>, (Ws<0>) → Wd<15> For byte operation: (Ws<7:1>) → Wd<6:0>, (Ws<0>) → Wd<7> 					
Status Affected:	N, Z					
Encoding:	1101	0011	0Bqq	qddd	dppp	ssss
Description:	Rotate the contents of the source register Ws one bit to the right and place the result in the destination register Wd. The Carry Flag bit is not affected. The 'B' bit selects byte or word operation. The 's' bits select the address of the source register. The 'd' bits select the address of the destination register. The 'p' bits select source address mode 2. The 'q' bits select destination address mode 2. See Table 1-5 and Table 1-6 for modifier addressing information. Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.					
Words:	1					
Cycles:	1					
Example:		RRNC	W5,W6	; Rotate right		
	Before Instruction					
	After Instruction					

TABLE 125 -- RRNCF: Rotate Right f (No Carry)

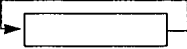
Syntax:	{label:}	RRNC{.b}	f	{,Ww}		
Operands:	f ∈ [0 ... 8191]					
Operation:	For word operation: (f<15:1>) → Dest<14:0>, (f<0>) → Dest<15> For byte operation: (f<7:1>) → Dest<6:0>, (f<0>) → Dest<7> 					
Status Affected:	N, Z					
Encoding:	1101	0111	0BDF	ffff	ffff	ffff
Description:	<p><u>Rotate the contents of the file register f one bit to the and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register. The carry flag bit is not affected.</u></p> <p><u>The 'B' bit selects byte or word operation.</u></p> <p><u>The 'D' bit selects the destination.</u></p> <p><u>The 'f' bits select the address of the file register.</u></p> <p><u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
Words:	1					
Cycles:	1					
Example:		RRNC	RAM135, Ww	; Rotate right		
	Before Instruction					
	After Instruction					

TABLE 126 -- SAC: Store Accumulator

Syntax:	{label:}	SAC	A,	Wnd,	[, Slit4]	
			B,	[Wnd],		
				[Wnd]++		
				[Wnd]--		
				[Wnd--],		
				[Wnd+Wb],		
				[Wnd+lit5]		
Operands:	<u>Wnd</u> \in [W0 ... W15]; <u>Wb</u> \in [W0 ... W15]; <u>lit5</u> \in [0 ... 31] <u>Slit4</u> \in [-8 ... +7]					
Operation:	<u>Shift_{Slit4}(ACC) (optional); (ACC[31:16]) \rightarrow Wnd</u>					
Status Affected:	<u>None</u>					
Encoding:	<u>1100</u>	<u>1100</u>	<u>Awww</u>	<u>wrrr</u>	<u>rh hh</u>	<u>ss ss</u>
Description:	<u>Optionally shift accumulator, then store truncated accumulator, ACC[31:16], to the destination effective address.</u> <u>The 'A' bits specify the source accumulator.</u> <u>The 's' bits specify the destination register Wnd.</u> <u>The 'h' bits select destination address mode 3.</u> <u>The 'w' bits specify the offset amount lit5 OR the offset register Wb.</u> <u>The 'r' bits encode the optional operand Slit4 which determines the amount of the accumulator preshift; if the operand Slit4 is absent, a 0 is encoded.</u> <u>See Table 1-7 for modifier addressing information.</u> <u>Note: Positive values of operand Slit4 represent arithmetic shift right.</u> <u>Negative values of operand Slit4 represent shift left.</u>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>SAC</u>	<u>A,W5</u>	<u>; Store Accumulator A</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 127 -- SCRATCH: Push Shadow Registers

<u>Syntax:</u>	<u>{label:}</u>	<u>PUSH.S</u>				
<u>Operands:</u>	<u>None</u>					
<u>Operation:</u>	<u>Push shadow registers. Shadowed registers include W0...W15 and STA-TUS.</u>					
<u>Status Affected:</u>	<u>None</u>					
<u>Encoding:</u>	<u>1111</u>	<u>1110</u>	<u>1010</u>	<u>0000</u>	<u>0000</u>	<u>0000</u>
<u>Description:</u>	<u>The contents of the primary registers are copied into the shadow registers.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1</u>					
<u>Example:</u>		<u>PUSH.S</u>		<u>: Push registers to shadows</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 128 -- SETF: Set or Ww

Syntax:	{label:}	SETM{.b}	f			
			<u>Ww</u>			
Operands:	<u>f ∈ [0 ... 8191]</u>					
Operation:	<u>0xFFFF → destination designated by D</u>					
Status Affected:	<u>None</u>					
Encoding:	<u>1110</u>	<u>1111</u>	<u>1BDf</u>	<u>ffff</u>	<u>ffff</u>	<u>ffff</u>
Description:	<p><u>Set the register designated by D: If the optional Ww is specified, D=0 and set Ww; otherwise, D=1 and set the file register.</u></p> <p><u>The 'B' bit selects byte or word operation.</u></p> <p><u>The 'f' bits select the address of the file register.</u></p> <p><u>The 'D' bit selects the destination.</u></p> <p><u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>SETM</u>	<u>345</u>	<u>; Set location 345</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 129 -- SFTAC: Arithmetic Shift Accumulator

Syntax:	{label:}	SFTAC	A,	Wb		
			B,			
Operands:	Wb ∈ [W0 ... W15]					
Operation:	Shift_(Wb)(ACC)					
Status Affected:	OA, OB, SA, SB					
Encoding:	1100	1000	A000	0000	0000	ssss
Description:	<p>Arithmetic shift of accumulator.</p> <p>The contents of Ws are used as the shift amount. Only the least significant 5 bits of the Ws are used. If Ws<4:0> is positive, the shift is a right shift by Ws<4:0> bits. If Ws<4:0> is negative, the shift is a left shift by -Ws<4:0> bits.</p> <p>The 'A' bit selects the accumulator for the result.</p> <p>The 's' bits select the address of the shift count register.</p>					
Words:	1					
Cycles:	1					
Example:		SFTAC	A,W5	; Shift Accumulator A right (W5) bits		
	Before Instruction					
	After Instruction					

TABLE 130 -- SFTACK: Arithmetic Shift Accumulator

Syntax:	{label:}	SFTAC	A,	Slit5		
			B,			
Operands:	Slit5 \in [-16 ... 15]					
Operation:	Shift_k(ACC)					
Status Affected:	OA, OB, SA, SB					
Encoding:	1100	1000	A100	0000	000k	kkkk
Description:	<p>Arithmetic shift of accumulator.</p> <p>The Slit5 is used as the shift amount. If Slit5 is positive, the shift is a right shift by Slit5 bits. If Slit5 is negative, the shift is a left shift by -Slit5 bits.</p> <p>The 'A' bit selects the accumulator for the result.</p> <p>The 'k' bits determine the number of bits to be shifted.</p>					
Words:	1					
Cycles:	1					
Example:		SFTAC	B,5	; Shift Accumulator B right five bits		
	Before Instruction					
	After Instruction					

TABLE 131 -- SL: Shift Left Ws


Syntax:	<u>{label:}</u>	<u>SL{.b}</u>	<u>Ws,</u>	<u>Wd</u>		
			<u>[Ws],</u>	<u>[Wd]</u>		
			<u>[Ws]++,</u>	<u>[Wd]++</u>		
			<u>[Ws]--,</u>	<u>[Wd]--</u>		
			<u>[Ws++]</u> ,	<u>[Wd++]</u>		
			<u>[Ws--]</u> ,	<u>[Wd--]</u>		
Operands:	<u>Ws</u> ∈ [W0 ... W15]; <u>Wd</u> ∈ [W0 ... W15]					
Operation:	<u>For word operation:</u> <u>(Ws<15>) → C, (Ws<14:0>) → Wd<15:1>, 0 → Wd<0></u> <u>For byte operation:</u> <u>(Ws<7>) → C, (Ws<6:0>) → Wd<7:1>, 0 → Wd<0></u> 					
Status Affected:	<u>C, N, OV, Z</u>					
Encoding:	<u>1101</u>	<u>0000</u>	<u>0Bqq</u>	<u>qddd</u>	<u>dppp</u>	<u>ssss</u>
Description:	<u>Shift the contents of the source register Ws one bit to the left and place the result in the destination register Wd. Shift '0' into the LSB of Wd. The Carry Flag is set if the MSB of Ws is '1'.</u> <u>The 'B' bit selects byte or word operation.</u> <u>The 's' bits select the address of the source register.</u> <u>The 'd' bits select the address of the destination register.</u> <u>The 'p' bits select source address mode 2.</u> <u>The 'q' bits select destination address mode 2.</u> <u>See Table 1-5 and Table 1-6 for modifier addressing information.</u> <u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>SL</u>	<u>W5,W6</u>	<u>; Shift left</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 132 -- SLEEP: Enter SLEEP mode

Syntax:	<u>{label:}</u>	<u>SLEEP</u>	<u>lit4</u>			
Operands:	<u>lit4</u> ∈ [0 ... 15]					
Operation:	<u>0 → WDT,</u> <u>0 → WDT prescaler count,</u> <u>1 → IQ</u> <u>0 → PD</u> <u>Enter sleep mode (lit4)</u>					
Status Affected:	<u>TO</u> <u>PD</u>					
Encoding:	<u>1111</u>	<u>1110</u>	<u>0100</u>	<u>0000</u>	<u>0000</u>	<u>kkkk</u>
Description:	<u>The power-down status bit</u> _{PD} <u> is cleared. Time-out status</u> _{TO} <u> is set. The Watchdog Timer and its prescaler are cleared. The processor is put into SLEEP mode selected by</u> <u>lit4.</u>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>SLEEP</u>	<u>0</u>	<u>; Turn off the device oscillator.</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 133 -- SLW: Shift Left by Wns

Syntax:	<u>{label:}</u>	<u>SL</u>	<u>Wb,</u>	<u>Wns,</u>	<u>Wnd</u>	
Operands:	<u>Wb ∈ [W0 ... W15]; Wns ∈ [W0 ... W15]; Wnd ∈ [W0 ... W15]</u>					
Operation:	<u>Wns<3:0>→Shift_Val</u> <u>0→Shift_In<39:16></u> <u>Wb<15:0>→Shift_In<15:0></u> <u>0→Shift_Out<39:16+Shift_Val></u> <u>Shift_In<15:0>→Shift_Out<15+Shift_Val:Shift_Val></u> <u>If Wns<4>==0: (less than 16)</u> <u>0→CARRY1<15:0></u> <u>Shift_Out<31:16>→CARRY0<15:0></u> <u>Shift_Out<15:0>→Wnd<15:0></u> <u>If Wns<4>==1: (16 or greater)</u> <u>Shift_Out<31:16>→CARRY1<15:0></u> <u>Shift_Out<15:0>→CARRY0<15:0></u> <u>0→Wnd<15:0></u>					
Status Affected:	<u>C,SZ,Z</u>					
Encoding:	<u>1101</u>	<u>1101</u>	<u>0www</u>	<u>wddd</u>	<u>d000</u>	<u>ssss</u>
Description:	<u>Shift left the contents of the source register Wb by Wns bits (up to 31 positions), placing the result in the destination register Wnd. Bits that are shifted beyond the leftmost position of the source are stored in the CARRY1 and CARRY0 registers.</u> <u>The Z and SZ bits will be set if the value placed in Wnd is zero and cleared otherwise. The C bit will be set if any of the bits shifted out were set (in other words, if the resultant CARRY is non-zero) and cleared otherwise.</u> <u>Note: This instruction operates in word mode only.</u>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					

TABLE 134 -- SRAC: Store Rounded Accumulator

Syntax:	{label:}	SAC.R	A,	Wnd,	[, Slit4]	
			B,	[Wnd],		
				[Wnd]++		
				[Wnd]--		
				[Wnd--],		
				[Wnd+Wb],		
				[Wnd+lit5]		
Operands:	Wnd \in [W0 ... W15]; Wb \in [W0 ... W15]; lit5 \in [0 ... 31] Slit4 \in [-8 ... +7]					
Operation:	Shift _{Slit4} (ACC) (optional); Round (ACC); (ACC[31:16]) \rightarrow Wnd					
Status Affected:	None					
Encoding:	1100	1101	Awww	wrrr	rh hh	ss ss
Description:	<p><u>Optionally shift accumulator, round and store convergent rounded accumulator, ACC, to the destination effective address.</u></p> <p><u>The 'A' bits specify the source accumulator.</u></p> <p><u>The 's' bits specify the destination register Wnd.</u></p> <p><u>The 'h' bits select destination address mode 3.</u></p> <p><u>The 'w' bits specify the offset amount lit5 OR the offset register Wb.</u></p> <p><u>The 'r' bits encode the optional operand Slit4 which determines the amount of the accumulator preshift; if the operand Slit4 is absent, a 0 is encoded.</u></p> <p><u>See Table 1-7 for modifier addressing information.</u></p> <p><u>Note: Positive values of operand Slit4 represent arithmetic shift right.</u> <u>Negative values of operand Slit4 represent shift left.</u></p>					
Words:	1					
Cycles:	1					
Example:		SAC.R	B,W5	; Store Rounded Accumulator		
	Before Instruction					
	After Instruction					

TABLE 135 -- SUB: Subtract Ws from Wb

Syntax:	<u>{label:}</u>	<u>SUB{.b}</u>	<u>Wb,</u>	<u>Ws,</u>	<u>Wd</u>	
				<u>[Ws],</u>	<u>[Wd]</u>	
				<u>[Ws]++,</u>	<u>[Wd]++</u>	
				<u>[Ws]--,</u>	<u>[Wd]--</u>	
				<u>[Ws++]</u> ,	<u>[Wd++]</u>	
				<u>[Ws--]</u> ,	<u>[Wd--]</u>	
Operands:	<u>Wb</u> \in [<u>W0 ... W15</u>]; <u>Ws</u> \in [<u>W0 ... W15</u>]; <u>Wd</u> \in [<u>W0 ... W15</u>]					
Operation:	<u>(Wb) - (Ws) \rightarrow Wd</u>					
Status Affected:	<u>C, DC, N, OV, Z</u>					
Encoding:	<u>0101</u>	<u>0www</u>	<u>wBqq</u>	<u>qddd</u>	<u>dppp</u>	<u>ssss</u>
Description:	<u>Subtract the contents of the source register Ws from the contents of the base register Wb and place the result in the destination register Wd.</u> <u>The 'B' bit selects byte or word operation.</u> <u>The 's' bits select the address of the source register.</u> <u>The 'w' bits select the address of the base register.</u> <u>The 'd' bits select the address of the destination register.</u> <u>The 'p' bits select source address mode 2.</u> <u>The 'q' bits select destination address mode 2.</u> <u>See Table 1-5 and Table 1-6 for modifier addressing information.</u> <u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>SUB</u>	<u>W5,W6,W7</u>	<u>; Subtract W5 from W6</u>		
	<u>Before Instruction</u> <u>After Instruction</u>					

TABLE 136 -- SUBAB: Subtract Accumulators

<u>Syntax:</u>	<u>{label:}</u>	<u>SUB</u>	<u>A</u>			
			<u>B</u>			
<u>Operands:</u>	<u>none</u>					
<u>Operation:</u>	<u>if (SUBAB A) then ACCA - ACCB → ACCA</u> <u>if (SUBAB B) then ACCB - ACCA → ACCB</u>					
<u>Status Affected:</u>	<u>OA, OB, SA, SB</u>					
<u>Encoding:</u>	<u>1100</u>	<u>1011</u>	<u>A011</u>	<u>0000</u>	<u>0000</u>	<u>0000</u>
<u>Description:</u>	<u>Subtract Accumulators and write results to selected accumulator.</u> <u>The 'A' bits specify the destination accumulator.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1</u>					
<u>Example:</u>		<u>SUB</u>	<u>B</u>	<u>; Subtract ACCA from ACCB, result to ACCB</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 137 -- SUBB: Subtract Ws from Wb with Borrow

Syntax:	{label:}	SUBB{.b}	Wb,	Ws,	Wd	
				[Ws],	[Wd]	
				[Ws]++,	[Wd]++	
				[Ws]--,	[Wd]--	
				[Ws++] ,	[Wd++]	
				[Ws--],	[Wd--]	
Operands:	Wb ∈ [W0 ... W15]; Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]					
Operation:	(Wb) - (Ws) - (C) → Wd					
Status Affected:	C, DC, N, OV, Z					
Encoding:	0101	1www	wBqq	qddd	dppp	ssss
Description:	<p>Subtract the contents of the source register Ws and the Carry flag from the contents of the base register Wb and place the result in the destination register Wd.</p> <p>The 'B' bit selects byte or word operation.</p> <p>The 's' bits select the address of the source register.</p> <p>The 'w' bits select the address of the base register.</p> <p>The 'd' bits select the address of the destination register.</p> <p>The 'p' bits select source address mode 2.</p> <p>The 'q' bits select destination address mode 2.</p> <p>See Table 1-5 and Table 1-6 for modifier addressing information.</p> <p>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</p>					
Words:	1					
Cycles:	1					
Example:			SUBB	W5,W6,W7	: Subtract	
	Before Instruction					
	After Instruction					

TABLE 138 -- SUBBFW: Subtract f and Carry bit from Ww

Syntax:	{label:}	SUBRB{.b}	f	{,Ww}		
Operands:	f ∈ [0 ... 8191]					
Operation:	(Ww) - (f) - (C) → destination designated by D					
Status Affected:	C, DC, N, OV, Z					
Encoding:	1011	1101	1BDf	ffff	ffff	ffff
Description:	<p>Subtract the contents of the file register and the carry bit from the contents of the working register and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.</p> <p>The 'B' bit selects byte or word operation.</p> <p>The 'D' bit selects the destination.</p> <p>The 'f' bits select the address of the file register.</p> <p>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</p>					
Words:	1					
Cycles:	1					
Example:		SUBRB		RAM135, Ww		; Subtract
	Before Instruction					
	After Instruction					

TABLE 139 -- SUBBLS: Subtract Short Literal from Wb with Borrow

<u>Syntax:</u>	<u>{label:}</u>	<u>SUBB{.b}</u>	<u>Wb,</u>	<u>lit5,</u>	<u>Wd</u>	
					<u>[Wd]</u>	
					<u>[Wd]++</u>	
					<u>[Wd]--</u>	
					<u>[Wd++]</u>	
					<u>[Wd--]</u>	
<u>Operands:</u>	<u>Wb ∈ [W0 ... W15]; lit5 ∈ [0 ... 31]; Wd ∈ [W0 ... W15]</u>					
<u>Operation:</u>	<u>(Wb) - lit5 - C → Wd</u>					
<u>Status Affected:</u>	<u>C, DC, N, OV, Z</u>					
<u>Encoding:</u>	<u>0101</u>	<u>1www</u>	<u>wBqq</u>	<u>qddd</u>	<u>d11k</u>	<u>kkkk</u>
<u>Description:</u>	<u>Subtract the literal operand and the Carry bit from the contents of the base register Wb and place the result in the destination register Wd.</u>					
	<u>The 'B' bit selects byte or word operation.</u>					
	<u>The 'w' bits select the address of the base register.</u>					
	<u>The 'k' bits provide the literal operand, a five-bit integer number.</u>					
	<u>The 'd' bits select the address of the destination register.</u>					
	<u>The 'q' bits select destination address mode 2.</u>					
	<u>See Table 1-6 for modifier addressing information.</u>					
	<u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1</u>					
<u>Example:</u>		<u>SUBB</u>	<u>W5,#12,W7</u>	<u>; Subtract</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 140 -- SUBBLW: Subtract Wn from Literal with Borrow

Syntax:	{label:}	SUBB{.b}	Slit10,	Wn		
Operands:	Slit10 \in [-512 ... 511]; Wn \in [W0 ... W15]					
Operation:	Slit10 - (Wn) - C \rightarrow Wn					
Status Affected:	C, DC, N, OV, Z					
Encoding:	1011	0001	1Bkk	kkkk	kkkk	dddd
Description:	<p>Subtract the literal operand and the Carry bit from the contents of the working register Wn and place the result in the working register Wn.</p> <p>The 'B' bit selects byte or word operation.</p> <p>The 'd' bits select the address of the working register.</p> <p>The 'k' bits specify the literal operand, a signed 10-bit number.</p> <p>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</p>					
Words:	1					
Cycles:	1					
Example:		SUBB	#123,W7	:	Subtract	
	Before Instruction					
	After Instruction					

TABLE 141 -- SUBBR: Subtract Wb from Ws with Borrow

Syntax:	{label:}	SUBBR{.b}	Wb,	Ws,	Wd	
				<u>[Ws],</u>	<u>[Wd]</u>	
				<u>[Ws]++,</u>	<u>[Wd]++</u>	
				<u>[Ws]--,</u>	<u>[Wd]--</u>	
				<u>[Ws++]</u> ,	<u>[Wd++]</u>	
				<u>[Ws--]</u> ,	<u>[Wd--]</u>	
Operands:	Wb ∈ [W0 ... W15]; Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]					
Operation:	(Ws) - (Wb) - (C) → Wd					
Status Affected:	C, DC, N, OV, Z					
Encoding:	<u>0001</u>	<u>1www</u>	<u>wBqq</u>	<u>qddd</u>	<u>dppp</u>	<u>ssss</u>
Description:	<p><u>Subtract the contents of the base register Wsb and the Carry flag from the contents of the source register Ws and place the result in the destination register Wd.</u></p> <p><u>The 'B' bit selects byte or word operation.</u></p> <p><u>The 's' bits select the address of the source register.</u></p> <p><u>The 'w' bits select the address of the base register.</u></p> <p><u>The 'd' bits select the address of the destination register.</u></p> <p><u>The 'p' bits select source address mode 2.</u></p> <p><u>The 'q' bits select destination address mode 2.</u></p> <p><u>See Table 1-5 and Table 1-6 for modifier addressing information.</u></p> <p><u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
Words:	1					
Cycles:	1					
Example:		SUBBR	W5,W6,W7	; Subtract W6 from W5 with borrow		
	Before Instruction					
	After Instruction					

TABLE 142 -- SUBBRLS: Subtract Wb from Short Literal with Borrow

Syntax:	{label:}	SUBBR{.b}	Wb,	lit5	Wd	
					[Wd]	
					[Wd]++	
					[Wd]--	
					[Wd++]	
					[Wd--]	
Operands:	Wb ∈ [W0 ... W15]; lit5 ∈ [0 ... 31]; Wd ∈ [W0 ... W15]					
Operation:	lit5 - (Wb) - (C) → Wd					
Status Affected:	C, DC, N, OV, Z					
Encoding:	0001	1www	wBqq	qddd	d11k	kkkk
Description:	<p><u>Subtract the contents of the base register Wb and the Carry flag from lit5 and place the result in the destination register Wd.</u></p> <p><u>The 'B' bit selects byte or word operation.</u></p> <p><u>The 'w' bits select the address of the base register.</u></p> <p><u>The 'k' bits provide the literal operand, a five-bit integer number.</u></p> <p><u>The 'd' bits select the address of the destination register.</u></p> <p><u>The 'q' bits select destination address mode 2.</u></p> <p><u>See Table 1-5 and Table 1-6 for modifier addressing information.</u></p> <p><u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
Words:	1					
Cycles:	1					
Example:		SUBBR	W5,#12,W7	; Subtract W5 from 12		
	Before Instruction					
	After Instruction					

TABLE 143 -- SUBBWF: Subtract Ww and Carry bit from f

Syntax:	{label:}	SUBB{.b}	f	{,Ww}		
Operands:	f ∈ [0 ... 8191]					
Operation:	(f) - (Ww) - (C) → destination designated by D					
Status Affected:	(C), DC, N, OV, Z					
Encoding:	1011	0101	1BDf	ffff	ffff	ffff
Description:	<p><u>Subtract the contents of the working register and the carry bit from the contents of the file register and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.</u></p> <p><u>The 'B' bit selects byte or word operation.</u></p> <p><u>The 'D' bit selects the destination.</u></p> <p><u>The 'f' bits select the address of the file register.</u></p> <p><u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
Words:	1					
Cycles:	1					
Example:		SUBB	RAM135, Ww	; Subtract		
	Before Instruction					
	After Instruction					

TABLE 144 -- SUBFW: Subtract f from Ww

Syntax:	<u>{label:}</u>	<u>SUBR{.b}</u>	<u>f</u>	<u>{,Ww}</u>		
Operands:	<u>f ∈ [0 ... 8191]</u>					
Operation:	<u>(Ww) - (f) → destination designated by D</u>					
Status Affected:	<u>C, DC, N, OV, Z</u>					
Encoding:	<u>1011</u>	<u>1101</u>	<u>0BDf</u>	<u>ffff</u>	<u>ffff</u>	<u>ffff</u>
Description:	<p><u>Subtract the contents of the file register from the contents of the working register and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.</u></p> <p><u>The 'B' bit selects byte or word operation.</u></p> <p><u>The 'D' bit selects the destination.</u></p> <p><u>The 'f' bits select the address of the file register.</u></p> <p><u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>SUBR</u>	<u>RAM135, ww</u>	<u>; Subtract</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 145 -- SUBLS: Subtract Short Literal from Wb

<u>Syntax:</u>	<u>{label:}</u>	<u>SUB{.b}</u>	<u>Wb,</u>	<u>lit5,</u>	<u>Wd</u>	
					<u>[Wd]</u>	
					<u>[Wd]++</u>	
					<u>[Wd]--</u>	
					<u>[Wd++]</u>	
					<u>[Wd--]</u>	
<u>Operands:</u>	<u>Wb ∈ [W0 ... W15]; lit5 ∈ [0 ... 31]; Wd ∈ [W0 ... W15]</u>					
<u>Operation:</u>	<u>(Wb) - lit5 → Wd</u>					
<u>Status Affected:</u>	<u>C, DC, N, OV, Z</u>					
<u>Encoding:</u>	<u>0101</u>	<u>0www</u>	<u>wBqq</u>	<u>qddd</u>	<u>d11k</u>	<u>kkkk</u>
<u>Description:</u>	<u>Subtract the literal operand from the contents of the base register Wb and place the result in the destination register Wd.</u> <u>The ‘B’ bit selects byte or word operation.</u> <u>The ‘w’ bits select the address of the base register.</u> <u>The ‘k’ bits provide the literal operand, a five-bit integer number.</u> <u>The ‘d’ bits select the address of the destination register.</u> <u>The ‘q’ bits select destination address mode 2.</u> <u>See Table 1-6 for modifier addressing information.</u> <u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1</u>					
<u>Example:</u>		<u>SUB</u>	<u>W5,#12,W7</u>	<u>: Subtract</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 146 -- SUBLW: Subtract Wn from Literal

Syntax:	{label:}	SUB{.b}	Slit10,	Wn		
Operands:	Slit10 ∈ [-512 ... 511]; Wn ∈ [W0 ... W15]					
Operation:	Slit10 - (Wn) → Wn					
Status Affected:	C, DC, N, OV, Z					
Encoding:	1011	0001	0Bkk	kkkk	kkkk	dddd
Description:	<p>Subtract the working register from the contents of the literal operand and place the result in the working register Wn.</p> <p>The 'B' bit selects byte or word operation.</p> <p>The 'd' bits select the address of the working register.</p> <p>The 'k' bits specify the literal operand, a signed 10-bit number.</p> <p><u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
Words:	1					
Cycles:	1					
Example:			SUB	#123,W7	: Subtract	
	Before Instruction					
	After Instruction					

TABLE 147 -- SUBR: Subtract Wb from Ws

<u>Syntax:</u>	<u>{label:}</u>	<u>SUBR{.b}</u>	<u>Wb,</u>	<u>Ws,</u>	<u>Wd</u>	
				<u>[Ws],</u>	<u>[Wd]</u>	
				<u>[Ws]++,</u>	<u>[Wd]++</u>	
				<u>[Ws]--,</u>	<u>[Wd]--</u>	
				<u>[Ws++]</u> ,	<u>[Wd++]</u>	
				<u>[Ws--]</u> ,	<u>[Wd--]</u>	
<u>Operands:</u>	<u>Wb ∈ [W0 ... W15]; Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]</u>					
<u>Operation:</u>	<u>(Ws) - (Wb) → Wd</u>					
<u>Status Affected:</u>	<u>C, DC, N, OV, Z</u>					
<u>Encoding:</u>	<u>0001</u>	<u>0www</u>	<u>wBqq</u>	<u>qddd</u>	<u>dppp</u>	<u>ssss</u>
<u>Description:</u>	<u>Subtract the contents of the base register Wb from the contents of the source register Ws and place the result in the destination register Wd.</u> <u>The 'B' bit selects byte or word operation.</u> <u>The 's' bits select the address of the source register.</u> <u>The 'w' bits select the address of the base register.</u> <u>The 'd' bits select the address of the destination register.</u> <u>The 'p' bits select source address mode 2.</u> <u>The 'q' bits select destination address mode 2.</u> <u>See Table 1-5 and Table 1-6 for modifier addressing information.</u> <u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u>					
<u>Words:</u>	<u>1</u>					
<u>Cycles:</u>	<u>1</u>					
<u>Example:</u>		<u>SUBR</u>	<u>W5,W6,W7</u>	<u>: Subtract W6 from W5</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 148 -- SUBRLS: Subtract Wb from Short Literal

Syntax:	{label:}	SUBR{.b}	Wb,	lit5	Wd	
					[Wd]	
					[Wd]++	
					[Wd]--	
					[Wd++]	
					[Wd--]	
Operands:	Wb ∈ [W0 ... W15]; lit5 ∈ [0 ... 31]; Wd ∈ [W0 ... W15]					
Operation:	lit5 - (Wb) → Wd					
Status Affected:	C, DC, N, OV, Z					
Encoding:	0001	0www	wBqq	qddd	d11k	kkkk
Description:	<p><u>Subtract the contents of the base register Wb from the lit5 and place the result in the destination register Wd.</u></p> <p><u>The 'B' bit selects byte or word operation.</u></p> <p><u>The 'w' bits select the address of the base register.</u></p> <p><u>The 'k' bits provide the literal operand, a five-bit integer number.</u></p> <p><u>The 'd' bits select the address of the destination register.</u></p> <p><u>The 'q' bits select destination address mode 2.</u></p> <p><u>See Table 1-5 and Table 1-6 for modifier addressing information.</u></p> <p><u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
Words:	1					
Cycles:	1					
Example:		SUBR		W5,#12,W7	; Subtract W5 from 12	
	Before Instruction					
	After Instruction					

TABLE 149 -- SUBWF: Subtract Ww from f

Syntax:	{label:}	SUB{.b}	f	{,Ww}		
Operands:	f ∈ [0 ... 8191]					
Operation:	(f) - (Ww) → destination designated by D					
Status Affected:	C, DC, N, OV, Z					
Encoding:	1011	0101	0BDf	ffff	ffff	ffff
Description:	<p>Subtract the contents of the working register from the contents of the file register and place the result in the destination designated by D. If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.</p> <p>The 'B' bit selects byte or word operation. The 'D' bit selects the destination. The 'f' bits select the address of the file register.</p> <p>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</p>					
Words:	1					
Cycles:	1					
Example:		SUB	RAM135, ww	; Subtract		
	Before Instruction					
	After Instruction					

TABLE 150 -- SWAP: Byte or Nibble Swap Wn

Syntax:	{label:}	SWAP	Wn			
Operands:	Wn ∈ [W0 ... W15]					
Operation:	If B=0; (Wn)<15:8> ↔ (Wn)<7:0> If B=1; (Wn)<7:4> ↔ (Wn)<3:0>					
Status Affected:	None					
Encoding:	1111	1101	1B00	0000	0000	ssss
Description:	If in word mode, byte swap Wn register. If in byte mode, nibble swap Wn register. Wn<15:8> are unaffected. The 'B' bit selects byte or word operation. The 's' bits select the address of the working register. Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.					
Words:	1					
Cycles:	1					
Example:		SWAP	W11	; Swap Bytes		
	Before Instruction					
	After Instruction					

TABLE 151 -- TBLRDH: Table Read High

Syntax:	{label:}	TBLRDH{.b}	[Ws],	Wd		
			[Ws]++,	[Wd]		
			[Ws]--,	[Wd]++		
			[Ws++] ,	[Wd]--		
			[Ws--],	[Wd++]		
				[Wd--]		
Operands:	Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]					
Operation:	In Word Mode: Program Mem [(PAGNUM),(Ws)]<23:16> → Wd <7:0> 0 → Wd <15:8> In Byte Mode: If LSB(Ws)=1, 0 → Wd<7:0> Else if LSB(Ws)=0, Program Mem [(PAGNUM),(Ws)] <23:16> → Wd<7:0>					
Status Affected:	None					
Encoding:	1011	1010	1Bqq	qddd	dppp	ssss
Description:	<p>This instruction is used to read the contents of program memory.</p> <p>The program memory address is calculated by concatenating the contents of the 8-bit Table Pointer (PAGNUM) register with the contents of the Ws register.</p> <p>Because the Ws value is always used as an address, the direct form of the first operand is invalid.</p> <p>The program memory word is stored in the location indicated by the Wd operand.</p> <p>For this instruction, the upper 8 bits of the program memory word (extended with '0's) are read.</p> <p>The 'B' bit selects byte or word operation.</p> <p>The 's' bits select the address of the source (address) register.</p> <p>The 'd' bits select the address of the destination (data) register.</p> <p>The 'p' bits select source address mode 2.</p> <p>The 'q' bits select destination address mode 2.</p> <p>Note: The extension .b in the instruction denotes a byte move rather than a word move. You may use a .w extension to denote a word move, but it is not required.</p>					
Words:	1					
Cycles:	2					
Example:		TBLRDH	W5, W6	; Read Program Memory High		
	Before Instruction After Instruction					

TABLE 152 -- TBLRDL: Table Read Low

Syntax:	{label:}	TBLRDL{.b}	[Ws],	Would		
			[Ws]++,	[Wd]		
			[Ws]--,	[Wd]++		
			[Ws++] ,	[Wd]--		
			[Ws--],	[Wd++]		
				[Wd--]		
Operands:	Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]					
Operation:	<p>In Word Mode: Program Mem [(PAGNUM),(Ws)] <15:0> → Wd</p> <p>In Byte Mode: If LSB(Ws)=1, Program Mem [(PAGNUM),(Ws)] <15:8> → Wd<7:0> Else if LSB(Ws)=0, Program Mem [(PAGNUM),(Ws)] <7:0> → Wd<7:0></p>					
Status Affected:	None					
Encoding:	1011	1010	0Bqg	qddd	dppp	ssss
Description:	<p>This instruction is used to read the contents of program memory.</p> <p>The program memory address is calculated by concatenating the contents of the 8-bit Table Pointer (PAGNUM) register with the contents of the Ws register.</p> <p>Because the Ws value is always used as an address, the direct form of the first operand is invalid.</p> <p>The program memory word is stored in the location indicated by the Wd operand.</p> <p>For this instruction, the lower 16 bits of the program memory word are read.</p> <p>The 'B' bit selects byte or word operation.</p> <p>The 's' bits select the address of the source (address) register.</p> <p>The 'd' bits select the address of the destination (data) register.</p> <p>The 'p' bits select source address mode 2.</p> <p>The 'q' bits select destination address mode 2.</p> <p>Note: The extension .b in the instruction denotes a byte move rather than a word move. You may use a .w extension to denote a word move, but it is not required.</p>					
Words:	1					
Cycles:	2					
Example:	TBLRDL	W5, W6	; Read Program Memory Low			
	Before Instruction					
	After Instruction					

TABLE 153 -- TBLWTH: Table Write High

Syntax:	{label:}	TBLWTH	Ws,	[Wd]		
			[Ws],	[Wd]++		
			[Ws]++,	[Wd]--		
			[Ws]--,	[Wd++]		
			[Ws++] ,	[Wd--],		
			[Ws--],			
Operands:	Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]					
Operation:	In Word Mode: (Ws)<7:0>→ Program Mem [(PAGNUM),(Wd)] <23:16> In Byte Mode: If LSB(Wd)=1, NOP Else if LSB(Wd)=0, Ws<7:0>→ Program Mem [(PAGNUM),(Wd)]<23:16>					
Status Affected:	None					
Encoding:	1011	1011	1Bqq	qddd	dppp	ssss
Description:	<p><u>This instruction is used to write the contents of Program Memory.</u></p> <p><u>The program memory address is calculated by concatenating the contents of the 8-bit Table Pointer (PAGNUM) register with the result of the Wd operand.</u></p> <p><u>Because the Wd value is always used as an address, the direct form of the second operand is invalid.</u></p> <p><u>The contents of the Ws operand are stored into program memory at the location indicated by the Wd operand.</u></p> <p><u>This instruction writes the upper 8 bits of the program memory word.</u></p> <p><u>The 'B' bit selects byte or word operation.</u></p> <p><u>The 's' bits select the address of the source (data) register.</u></p> <p><u>The 'd' bits select the address of the destination (address) register.</u></p> <p><u>The 'p' bits select source address mode 2.</u></p> <p><u>The 'q' bits select destination address mode 2.</u></p> <p><u>Note: The extension .b in the instruction denotes a byte move rather than a word move. You may use a .w extension to denote a word move, but it is not required.</u></p>					
Words:	1					
Cycles:	2					
Example:	TBLWTH		W5, W6		; Load Program Memory High	
	Before Instruction After Instruction					

TABLE 154 -- TBLWTL: Table Write Low

Syntax:	{label:}	TBLWTL{.b}	Ws,	[Wd]		
			[Ws],	[Wd]++		
			[Ws]++,	[Wd]--		
			[Ws]--,	[Wd++]		
			[Ws++] ,	[Wd--],		
			[Ws--],			
Operands:	Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15] S ∈ [0, 1] (default = 0)					
Operation:	<u>In Word Mode:</u> (Ws) → Program Mem [(PAGNUM),(Wd)] <15:0> <u>In Byte Mode:</u> If LSB(Ws)=1, Ws<7:0> → Program Mem [(PAGNUM),(Wd)] <15:8> Else if LSB(Wd)=0, Ws<7:0> → Program Mem [(PAGNUM),(Wd)] <7:0>					
Status Affected:	None					
Encoding:	<u>1011</u>	<u>1011</u>	<u>0Bqq</u>	<u>qddd</u>	<u>dppp</u>	<u>ssss</u>
Description:	<u>This instruction is used to program the contents of Program Memory.</u> <u>The program memory address is calculated by concatenating the contents of the 8-bit Table Pointer (PAGNUM) register with the result of the Wd operand.</u> <u>Because the Wd value is always used as an address, the direct form of the second operand is invalid.</u> <u>The contents of the Ws operand are stored into program memory at the location indicated by the Wd operand.</u> <u>This instruction writes the lower 16 bits of the program memory word.</u> <u>The 'B' bit selects byte or word operation.</u> <u>The 's' bits select the address of the source (data) register.</u> <u>The 'd' bits select the address of the destination (address) register.</u> <u>The 'p' bits select source address mode 2.</u> <u>The 'q' bits select destination address mode 2</u> <u>Note: The extension .b in the instruction denotes a byte move rather than a word move. You may use a .w extension to denote a word move, but it is not required.</u>					
Words:	<u>1</u>					
Cycles:	<u>2</u>					
Example:		<u>TBLWTL</u>	<u>W5, W6</u>	<u>; Load Program Memory Low</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 155 -- ULNK: De-allocate Stack Frame

Syntax:	{label:}	ULNK				
Operands:	None					
Operation:	W14 → W15; [W15++] → W14					
Status Affected:	None					
Encoding:	<u>1111</u>	<u>1010</u>	<u>1000</u>	<u>0000</u>	<u>0000</u>	<u>0000</u>
Description:	<u>This instruction de-allocates a stack frame and adjusts the stack pointer and frame pointer.</u>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>ULNK</u>		<u>; Deallocate stack frame</u>		

TABLE 156 -- TRAP: Trap to vector(lit1) with lit16

Syntax:	<u>{label:}</u>	<u>TRAP</u>	<u>lit1,</u>	<u>lit16</u>		
Operands:	<u>lit1</u> \in [0,1]; <u>lit16</u> \in [0 ... 65535]					
Operation:	<u>(PC) +2 \rightarrow PC,</u> <u>(PC<15:0>) \rightarrow TOS,</u> <u>(W15)+2 \rightarrow W15</u> <u>(PC<23:16>) \rightarrow TOS,</u> <u>(W15)+2 \rightarrow W15</u> <u>Vector(lit1) \rightarrow PC;</u> <u>lit16 \rightarrow TOS</u>					
Status Affected:	<u>None</u>					
Encoding:	<u>0000</u>	<u>101n</u>	<u>kkkk</u>	<u>kkkk</u>	<u>kkkk</u>	<u>kkkk</u>
Description:	<u>This instruction allows instruction expansion. The instruction will call a vector location with the lit16 value pushed onto the stack.</u>					
Words:	<u>1</u>					
Cycles:	<u>2</u>					
Example:		<u>TRAP</u>	<u>#0,#0x5A5A</u>			

TABLE 157 -- XOR: Exclusive or Wb and Ws

Syntax:	{label:}	XOR{.b}	Wb,	Ws,	Wd	
				[Ws],	[Wd]	
				[Ws]++,	[Wd]++	
				[Ws]--,	[Wd]--	
				[Ws++]	[Wd++]	
				[Ws--]	[Wd--]	
Operands:	Wb ∈ [W0 ... W15]; Ws ∈ [W0 ... W15]; Wd ∈ [W0 ... W15]					
Operation:	(Wb).XOR.(Ws) → Wd					
Status Affected:	N, Z					
Encoding:	0110	1www	wBqq	qddd	dppp	ssss
Description:	<p>Compute Exclusive OR of the contents of the source register Ws and the contents of the base register⁴ Wb and place the result in the destination register Wd.</p> <p>The 'B' bit selects byte or word operation.</p> <p>The 's' bits select the address of the source register.</p> <p>The 'w' bits select the address of the base register.</p> <p>The 'd' bits select the address of the destination register.</p> <p>The 'p' bits select source address mode 2.</p> <p>The 'q' bits select destination address mode 2.</p> <p>See Table 1-5 and Table 1-6 for modifier addressing information.</p> <p><u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
Words:	1					
Cycles:	1					
Example:		XOR	W5,W6,W7	; Xor		
	Before Instruction					
	After Instruction					

TABLE 158 -- XORLS: Exclusive Or Wb and Short Literal

Syntax:	{label:}	XOR{.b}	Wb,	lit5,	Wd	
					[Wd]	
					[Wd]++	
					[Wd]--	
					[Wd++]	
					[Wd--]	
Operands:	Wb ∈ [W0 ... W15]; lit5 ∈ [0 ... 31]; Wd ∈ [W0 ... W15]					
Operation:	(Wb).XOR.lit5 → Wd					
Status Affected:	N, Z					
Encoding:	0110	1www	wBqq	qddd	d11k	kkkk
Description:	<p>Compute the Exclusive Or of the contents of the base register Wb and the literal operand and place the result in the destination register Wd.</p> <p>The 'B' bit selects byte or word operation.</p> <p>The 'w' bits select the address of the base register.</p> <p>The 'k' bits provide the literal operand, a five-bit integer number.</p> <p>The 'd' bits select the address of the destination register.</p> <p>The 'q' bits select destination address mode 2.</p> <p>See Table 1-6 for modifier addressing information.</p> <p><u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
Words:	1					
Cycles:	1					
Example:			XOR	W5,#12,W7	: Exclusive Or	
	Before Instruction					
	After Instruction					

TABLE 159 -- XORLW: Exclusive Or Literal and Wn

Syntax:	<u>{label:}</u>	<u>XOR{.b}</u>	<u>Slit10,</u>	<u>Wn</u>		
Operands:	<u>Slit10</u> ∈ [-512 ... 511]; <u>Wn</u> ∈ [W0 ... W15]					
Operation:	<u>Slit10.XOR.(Wn) → Wn</u>					
Status Affected:	<u>N, Z</u>					
Encoding:	<u>1011</u>	<u>0010</u>	<u>1Bkk</u>	<u>kkkk</u>	<u>kkkk</u>	<u>dddd</u>
Description:	<p><u>Compute the Exclusive Or of the literal operand and the contents of the working register Wn and place the result in the working register Wn.</u></p> <p><u>The 'B' bit selects byte or word operation.</u></p> <p><u>The 'd' bits select the address of the working register.</u></p> <p><u>The 'k' bits specify the literal operand, a signed 10-bit number.</u></p> <p><u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>XOR</u>	<u>#123,W7</u>	<u>: Exclusive Or</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

TABLE 160 -- XORWF: Exclusive Or f and Ww

Syntax:	<u>{label:}</u>	<u>ADD{.b}</u>	<u>f</u>	<u>{,Ww}</u>		
Operands:	<u>f ∈ [0 ... 8191]</u>					
Operation:	<u>(f).XOR.(Ww) → destination designated by D</u>					
Status Affected:	<u>N, Z</u>					
Encoding:	<u>1011</u>	<u>0110</u>	<u>1B Df</u>	<u>ffff</u>	<u>ffff</u>	<u>ffff</u>
Description:	<p><u>Compute the XOR of the contents of the working register and the contents of the file register and place the result in the destination designated by D: If the optional Ww is specified, D=0 and store result in Ww; otherwise, D=1 and store result in the file register.</u></p> <p><u>The 'B' bit selects byte or word operation.</u></p> <p><u>The 'D' bit selects the destination.</u></p> <p><u>The 'f' bits select the address of the file register.</u></p> <p><u>Note: The extension .b in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.</u></p>					
Words:	<u>1</u>					
Cycles:	<u>1</u>					
Example:		<u>XOR</u>	<u>RAM135, Ww</u>	<u>; Exclusive Or</u>		
	<u>Before Instruction</u>					
	<u>After Instruction</u>					

INSTRUCTION OPERATION DETAILS

[0222] An explanation of the instruction operation details are enhanced by reference to several figures, specifically Figures 6-23 and 174.

Implied W register Utilization

[0223] Certain W registers have implied utilization in the instruction set. W0-W3 are used as the operands for DSP instructions. W4-W7 are used as the prefetch addresses for DSP instructions. W14 is the frame pointer utilized by the LNK and ULNK instructions. W15 acts as the stack pointer.

TABLE 161 -- Implied W Register Utilization

<u>Register</u>	
<u>W0</u>	<u>MAC operand; Default Ww</u>
<u>W1</u>	<u>MAC operand</u>
<u>W2</u>	<u>MAC operand; MULWF product LSB</u>
<u>W3</u>	<u>MAC operand; MULWF product MSB</u>
<u>W4</u>	<u>MAC prefetch address</u>
<u>W5</u>	<u>MAC prefetch address</u>
<u>W6</u>	<u>MAC prefetch address</u>
<u>W7</u>	<u>MAC prefetch address</u>
<u>W8</u>	<u>MAC prefetch offset</u>
<u>W9</u>	<u>MAC write back address</u>
<u>W10</u>	
<u>W11</u>	
<u>W12</u>	
<u>W13</u>	
<u>W14</u>	<u>Frame Pointer</u>
<u>W15</u>	<u>Stack Pointer</u>

Default Ww

[0224] W0 serves as the default Ww register for file register instructions. In this capacity, Ww acts as the W register in C16 and C18 compatible instructions.

Byte Operations

[0225] When a byte is moved into a W register, the byte is written into the LSbyte of the register and the MSbyte is left alone. Byte operations on the registers will operate on

the LSbyte of the register. The MSbyte of the register is left alone. For byte operations, the status flags will be adjusted to respond to the <7:0> bits of the register. For example, the carry bit will originate from ALU<7>. When a byte is moved from a W register, the source is the LSbyte and it overwrites the target byte in the memory. Other bytes are not affected.

Byte Operations in Bit Instructions - W Registers

[0226] The Bit operation instructions that use the W registers can address bytes or words without the requirement for a B bit. These instructions include BCLR, BSET, BSW.C, BSW.Z, BTG, BTST.C, BTST.Z, BTSTS.C, BTSTS.Z, BTST.C and BTST.Z. This works by making the bit field selection look at the LSB of the word or byte being addressed by the W register. If the address of the word or byte LSB is one, then zero that LSB and set the MSB of the bit selection field.

W0 = 1000

W1 = 1001

BCLR W0,#5 ; Clear 5th bit in word 1000

BCLR W0,#13 ; Clear 13th bit in word 1000

BCLR W1,#5 ; Clear 5th bit in byte 1001, same as
clear 13th bit in word 1000.

BCLR W1,#13 ; Invalid, same as
clear 13th bit in word 1000.

Using 10-bit literals

[0227] The instructions that have 10-bit literals have byte and word modes. For byte instructions, the literal is truncated at 8 bits. If the user specifies a signed value {-128... -1}, the truncated 2's compliment is coded. Unsigned values may range from {0 ... 255}. For word instructions, the literal is sign extended to 16-bits.

TABLE 162 -- 10-BIT LITERAL CODING

<u>Literal Value</u>	<u>If B=0 (Word) kk kkkk kkkk</u>	<u>If B=1 (Byte) kk kkkk kkkk</u>
<u>-512</u>	<u>10 0000 0000</u>	<u>n/a</u>
<u>-511</u>	<u>10 0000 0001</u>	<u>n/a</u>

<u>-129</u>	<u>11 0111 1111</u>	<u>n/a</u>
<u>-128</u>	<u>11 1000 0000</u>	<u>11 1000 0000</u>
<u>-2</u>	<u>11 1111 1110</u>	<u>11 1111 1110</u>
<u>-1</u>	<u>11 1111 1111</u>	<u>11 1111 1111</u>
<u>0</u>	<u>00 0000 0000</u>	<u>00 0000 0000</u>
<u>1</u>	<u>00 0000 0001</u>	<u>00 0000 0001</u>
<u>2</u>	<u>00 0000 0010</u>	<u>00 0000 0010</u>
<u>127</u>	<u>00 0111 1111</u>	<u>00 0111 1111</u>
<u>128</u>	<u>00 1000 0000</u>	<u>00 1000 0000</u>
<u>255</u>	<u>00 1111 1111</u>	<u>00 1111 1111</u>
<u>256</u>	<u>01 0000 0000</u>	<u>n/a</u>
<u>511</u>	<u>11 1111 1111</u>	<u>n/a</u>

Program Memory Addressing

[0228] Program memory contains a user space and a test space. The most significant bit (PMA<23>) of the program memory address selects user/test space. The least significant bit (PMA<0>) selects a byte for data addressing and table addressing modes.

[0229] Program memory addresses coded into instructions are coded in a lit23 or Slit16 format. The lit23 format encodes a direct address that represents PMA<22:0>. PMA<23> is not valid user space and is not encoded. The Slit16 format encodes an instruction count offset. The offset is added to the PC to generate the next address. The Slit16 format does not encode the PMA<0> bit as it represents an instruction count. The Slit16<15> bit is sign extended when added to the PC.

[0230] Figure 6 shows a block diagram illustrating a Program Memory Addressing Scheme. Figure 7 shows a block diagram illustrating a "CALL lit23" Map to the Program Counter. Figure 8 shows a block diagram illustrating a "BRA SLIT16" Map to the Program Counter. Figure 9 shows a block diagram illustrating a "GOTO Wn" Map to the Program Counter. Figure 10 shows a block diagram illustrating a "BRA Wn" Map to the Program Counter.

Shadows

[0231] Shadow registers are 1 level deep mini-stack registers attached to several key user registers. A PUSH.S will copy the user registers to the shadows and a POP.S will copy the shadows back to the user registers. Shadow registers are attached to W0...W15, the STATUS register, and the LCR,LSR,LER registers used by DO and REPEAT instructions.

MAC

[0232] The MAC instruction is a pipelined instruction. The first pipeline stage generates the effective addresses of the X and Y data and fetches the X and Y data. The second pipeline stage computes the multiply and accumulate, storing the results into the accumulator.

FORMS

[0233] The MAC instruction, and variants, can have several formats. Fundamentally, it must specify a target accumulator and a multiplicand and multiplier (ACC=X*Y). For Example:

MAC A,W0*W1

[0234] The MAC can also specify a prefetch for the next X or Y operand. The assembler can discriminate the X or Y prefetch based on the register used as the indirect address. [W4] or [W5] specifies the X prefetch and [W6] or [W7] specifies the Y prefetch. If a prefetch is specified, it must have a prefetch destination register. Legal forms of prefetch include:

MAC A,W0*W1,W0,[W4] ;X only

MAC A,W0*W1,W1,[W6] ;Y only

MAC A,W0*W1,W0,[W4],W1,[W6] ;X,Y

[0235] A write back can be specified. The write back uses the W9 register as the destination address. In this way, the assembler can discern the write back option.

MAC A,W0*W1,[W9] ;WBack only
MAC A,W0*W1,W0,[W6],W9 ;Y,WBack
MAC A,W0*W1,W0,[W4],[W9] ;X,Wback
MAC A,W0*W1,W0,[W4],W1,[W6],W9

SQUARING OPERATIONS

[0236] Squaring in the DSP engine is done with the square PLA opcodes. These are variants of the MAC and MPY opcodes.

For Example:

MAC B,W0*W0,W0,[W4],W1,[W6]+=2,W9

[0237] This instruction will multiply W0 time W0 and write the result in ACCB while doing the prefetch and write back. The assembler can tell that a MAC or MPY should translate to SQRAC or SQR instructions by finding the Wm*Wm format.

File Registers

[0238] File registers include parts of user RAM area and the Special Function Registers (SFR). The file register space is 8192 bytes. The file registers are directly addressable using the f field in the file register instructions.

[0239] All data addresses are byte addresses. When using byte instructions, the bytes are addressed directly. When using word instructions, the address must be word aligned. The least significant address bit must be 0. Figure 11 shows a block diagram illustrating a Data Alignment in Memory.

Carry and Borrow in PIC instructions

[0240] The PIC uses one unified carry and borrow bit, the C bit in the status register. The following examples show the functionality of the carry/borrow.

[0241] If a normal add generates a carry out of the 15th bit, the carry bit is set.

ADD 1 + 65535

1 =	0000 0000 0000 0001
+ 65535 =	1111 1111 1111 1111
-----	-----
0 =	0000 0000 0000 0000
C =	1
Z =	1
N =	0
OV =	0

[0242] An add carry will use the carry bit as an additional input. If the add generates a carry out of the 15th bit, the carry bit is set.

ADDC 1 + 65535, no carry in

1 =	0000 0000 0000 0001
+ 65535 =	1111 1111 1111 1111
C =	0
-----	-----
0 =	0000 0000 0000 0000
C =	1
Z =	1
N =	0
OV =	0

ADDC 1 + 65535, carry in

1 =	0000 0000 0000 0001
+ 65535 =	1111 1111 1111 1111
C =	1
-----	-----
0 =	0000 0000 0000 0001
C =	1
Z =	0
N =	0
OV =	0

[0243] A subtract instruction inverts the bits of the subtrahend, forces the carry in to 1 and does an add. This has the effect of generating the 2's compliment of the subtrahend. If the add generates a carry out of the 15th bit, the carry bit is set. However, in the case of a subtract, the carry bit is viewed as a BORROW bit. So a 1 in the carry bit indicates no borrow. A 0 in the carry bit indicates a borrow.

[0244] Subtracting 3 - 2 generates no borrow, so the C bit is 1.

```

SUB 3 - 2
  3 = 0000 0000 0000 0011
+ not 2 = 1111 1111 1111 1101
  C = 1
  ---
  1 = 0000 0000 0000 0001
  C = 1
  Z = 0
  N = 0
  OV = 0

```

[0245] Subtracting 3 - 3 generates no borrow, so the C bit is 1. The Z bit indicates a zero result.

```

SUB 3 - 3
  3 = 0000 0000 0000 0011
+ not 3 = 1111 1111 1111 1100
  C = 1
  ---
  0 = 0000 0000 0000 0000
  C = 1
  Z = 1
  N = 0
  OV = 0

```

[0246] Subtracting 2 - 3 generates a borrow, so the C bit is 0. The N bit indicates a negative result.

```

SUB 2 - 3
  2 = 0000 0000 0000 0010
+ not 3 = 1111 1111 1111 1100
  C = 1
  ---
 -1 = 1111 1111 1111 1111
  C = 0
  Z = 0
  N = 1
  OV = 0

```

[0247] A subtract with borrow instruction inverts the bits of the subtrahend, leaves the carry at its previous state and does an add. This has the effect of generating the 2's compliment of the subtrahend while inputting a BORROW bit.

[0248] Subtract/borrow 3 - 2 with no borrow in generates no borrow, so the C bit is

1.

SUBB 3 - 2, no borrow in

3 = 0000 0000 0000 0011

+ not 2 = 1111 1111 1111 1101

C = 1

1 = 0000 0000 0000 0001

C = 1

Z = 0

N = 0

OV = 0

[0249] Subtract/borrow 3 - 2 with borrow in generates no borrow, so the C bit is 1.

The result is 0, so the Z bit is set.

SUBB 3 - 2, borrow in

3 = 0000 0000 0000 0011

+ not 2 = 1111 1111 1111 1101

C = 0

0 = 0000 0000 0000 0000

C = 1

Z = 1

N = 0

$$\underline{\underline{OV = 0}}$$

[0250] Subtract/borrow 2 - 3 with borrow in generates a borrow, so the C bit is 0.

The N bit indicates a negative result.

```

SUBB 2 - 3, borrow in
  2 = 0000 0000 0000 0010
+ not 3 = 1111 1111 1111 1100
-----
  C = 0
  -2 = 1111 1111 1111 1110
  C = 0
  Z = 0
  N = 1
  OV = 0

```

Overflow Conditions

[0251] When doing 2's compliment mathematics, the OV flag indicates an overflow.

When doing multi-word math, the overflow is ignored until the most significant operation.

```

SUB 32760 - -32768
  32760 = 0111 1111 1111 1000
+ not 32768 = 0111 1111 1111 1111
-----
  C = 1
  -8 = 1111 1111 1111 1000
  C = 0
  Z = 0
  N = 1
  OV = 1

```

```

SUB -32760 - 32767
 - 32760 = 1000 0000 0000 1000
+ not 32767 = 1000 0000 0000 0001
-----
  C = 1
  10 = 0000 0000 0000 1010
  C = 1
  Z = 0
  N = 0
  OV = 1

```

Branch Conditions

[0252] Conditional branch instructions are valid after compare or subtract instructions. The compare is minuend-subtrahend and the condition tests are in the same order. For example, BGT will be true if the minuend is greater than the subtrahend or (minuend > subtrahend).

TABLE 163 -- Table 163: Branch conditions

Instruction		Status Test
BRA	C,Slit16	C
BRA	GE,Slit16	(N&&OV) (N&&OV)
BRA	GEU,Slit16	C
BRA	GT,Slit16	(Z&&N&&OV) (Z&&N&&OV)
BRA	GTU,Slit16	C&&Z
BRA	LE,Slit16	Z (N&&OV) (N&&OV)
BRA	LEU,Slit16	C Z
BRA	LT,Slit16	(N&&OV) (N&&OV)
BRA	LTU,Slit16	C
BRA	N,Slit16	N
BRA	NC,Slit16	C
BRA	NN,Slit16	N
BRA	NOV,Slit16	OV
BRA	NZ,Slit16	Z
BRA	OV,Slit16	OV
BRA	Z,Slit16	Z

TABLE 164 -- EXAMPLE BRANCH COMPARISON TESTS

Minu	Subtr	C	Z	N	OV	LT	LTU	LE	LEU	GE	GEU	GT	GTU
3	2	1	0	0	0	0	0	0	0	1	1	1	1
3	3	1	1	0	0	0	0	1	1	1	1	0	0
2	3	0	0	1	0	1	1	1	1	0	0	0	0
32760	-32768	0	0	1	1	0	1	0	1	1	0	1	0
-or-	32768												
32760													
-32760	32767	1	0	0	1	1	0	1	0	0	1	0	1
-or-	32767												
32776													

Stack operation

[0253] The dsPIC stack is a software stack implemented in user RAM area. While the device has provisions to allow pointer manipulation on any of the 16 W registers, W15 is the assumed stack pointer.

[0254] The stack starts at lower memory and grows towards high memory. The stack pointer points to the next available location. The stack pointer is manipulated with the source and destination addressing modes as shown in Table 192 and Table 193. With respect to Figures 173a-d, a push is `MOV W0, [W15]++` and a pop is `MOV [W15--], W0`.

[0255] Figure 173a shows a block diagram illustrating a stack pointer at initialization. Figure 173b shows a block diagram illustrating a stack pointer after a PUSH operation (`MOV W0, [W15]++`). Figure 173c shows a block diagram illustrating a stack pointer after a PUSH operation (`MOV W1, [W15]++`). Figure 173d shows a block diagram illustrating a stack pointer after a POP operation (`MOV [W15--], W3`).

Multi-word Move operations

[0256] The multi-word move instructions manipulated with the source and destination addressing modes as shown in Table 192 and Table 193. Figure 12 shows a block diagram illustrating a MOV.D operation. Figure 13 shows a block diagram illustrating a MOV.Q operation.

TABLE 165 -- STDW OPERATION

Instr. Cycle	Ws	[Ws]	[Ws++]	[Ws--]	[Ws]++	[Ws]--
Q1	W(nd)? Ws	Ws=Ws+2	Ws=Ws+6	Ws=Ws-2	W(nd)? (Ws) Ws=Ws+2	W(nd)? (Ws) Ws=Ws+2
Q2		W(nd+1)? (Ws)	W(nd+1)? (Ws)	W(nd+1)? (Ws)		
Q3	W(nd+1)? W(s+1)	Ws=Ws-2	Ws=Ws-2	Ws=Ws-2	W(nd+1)? (Ws) Ws=Ws+2	W(nd+1)? (Ws) Ws=Ws-6
Q4		W(nd)? (Ws)	W(nd)? (Ws)	W(nd)? (Ws)		

TABLE 166 -- LDDW OPERATION

Instr. Cycle	Ws	[Ws]	[Ws++]	[Ws--]	[Ws]++	[Ws]--
Q1	Ws? W(nd)	Ws=Ws+2	Ws=Ws+6	Ws=Ws-2	(Ws)? W(nd) Ws=Ws+2	(Ws)? W(nd) Ws=Ws+2
Q2		(Ws)? W(nd+1)⊆	(Ws)? W(nd+1)	(Ws)? W(nd+1)		
Q3	W(s+1)? W(nd+1)	Ws=Ws-2	Ws=Ws-2	Ws=Ws-2	(Ws)? W(nd+1) Ws=Ws+2	(Ws)? W(nd+1) Ws=Ws-6
Q4		(Ws)? W(nd)	(Ws)? W(nd)	(Ws)? W(nd)		

TABLE 167 -- STQW OPERATION

Instr. Cycle	Ws	[Ws]	[Ws++]	[Ws--]	[Ws]++	[Ws]--
Q1		Ws=Ws+6	Ws=Ws+14	Ws=Ws-2		
Q2	W(nd)? Ws	W(nd+3)? (Ws)	W(nd+3)? (Ws)	W(nd+3)? (Ws)	W(nd)? (Ws) Ws=Ws+2	W(nd)? (Ws) Ws=Ws+2
Q3		Ws=Ws-2	Ws=Ws-2	Ws=Ws-2		
Q4	W(nd+1)? W(s+1)	W(nd+2)? (Ws)	W(nd+2)? (Ws)	W(nd+2)? (Ws)	W(nd+1)? (Ws) Ws=Ws+2	W(nd+1)? (Ws) Ws=Ws+2
Q1		Ws=Ws-2	Ws=Ws-2	Ws=Ws-2		
Q2	W(nd+2)? W(s+2)	W(nd+1)? (Ws)	W(nd+1)? (Ws)	W(nd+1)? (Ws)	W(nd+2)? (Ws) Ws=Ws+2	W(nd+2)? (Ws) Ws=Ws+2
Q3		Ws=Ws-2	Ws=Ws-2	Ws=Ws-2		
Q4	W(nd+3)? W(s+3)	W(nd)? (Ws)	W(nd)? (Ws)	W(nd)? (Ws)	W(nd+3)? (Ws) Ws=Ws+2	W(nd+3)? (Ws) Ws=Ws-14

TABLE 168 -- LDQW OPERATION

Instr. Cycle	Ws	[Ws]	[Ws++]	[Ws--]	[Ws]++	[Ws]--
Q1		Ws=Ws+6	Ws=Ws+14	Ws=Ws-2		
Q2	Ws? W(nd)	W(s+3)? W(nd+3)	W(s+3)? W(nd+3)	W(s+3)? W(nd+3)	W(nd)? (Ws) Ws=Ws+2	W(nd)? (Ws) Ws=Ws+2
Q3		Ws=Ws-2	Ws=Ws-2	Ws=Ws-2		
Q4	W(s+1)? W(nd+1)	W(s+2)? W(nd+2)	W(s+2)? W(nd+2)	W(s+2)? W(nd+2)	W(s+1)? W(nd+1) Ws=Ws+2	W(s+1)? W(nd+1) Ws=Ws+2
Q1		Ws=Ws-2	Ws=Ws-2	Ws=Ws-2		
Q2	W(s+2)? W(nd+2)	W(s+1)? W(nd+1)	W(s+1)? W(nd+1)	W(s+1)? W(nd+1)	W(s+2)? W(nd+2) Ws=Ws+2	W(s+2)? W(nd+2) Ws=Ws+2
Q3		Ws=Ws-2	Ws=Ws-2	Ws=Ws-2		
Q4	W(s+3)? W(nd+3)	Ws? W(nd)	Ws? W(nd)	Ws? W(nd)	W(s+3)? W(nd+3) Ws=Ws+2	W(s+3)? W(nd+3) Ws=Ws-14

Link and Unlink Instructions

[0257] The link and unlink instructions assume that W15 is a stack pointer and W14 is a frame pointer. The link instruction is used during a calling sequence. Figure 14 shows a block diagram illustrating a stack at the beginning of a calling sequence. Before calling the subroutine, the parameters of the routine are pushed on the stack.

```

PUSH W0           ;Push parameter 1
PUSH W1           ;Push parameter n-1
PUSH W2           ;Push parameter n
CALL SUBR

```

[0258] Figure 15 shows a block diagram illustrating a stack at the entry to a routine.

```

SUBR:      LNK      2      ;Allocate 2 words

```

[0259] The LNK instruction will push the calling routines FP onto the stack. The new FP will be set to point to the current stack pointer. Then the literal is subtracted from the stack pointer which reserves the amount of memory allocated. Figure 16 shows a block diagram illustrating a stack after a LNK instruction.

[0260] Inside of the routine, the stack is used to save values. [W14+n] will access the Temp locations used by the routine. [W14-n] is used to access the parameters.

[0261] At the end of the routine, the ULNK instruction will copy the FP to the stack pointer then POP the callers FP back to the FP.

ULNK ;De-allocate frame

This returns the stack back to the state in Figure 15.

[0262] A return instruction will return to the caller. The caller is responsible for removing the parameters from the stack.

RETURN
POP W2 ;Unload parameter 1
POP W1 ;Unload parameter n-1
POP W0 ;Unload parameter n

This returns the stack back to the state in Figure 14.

Multi-word Shift Instructions

[0263] The CARRY1 and CARRY0 registers hold the temporary values of the shift. Figure 17 shows a block diagram illustrating a Multi-Word Left Shift by 4 Instruction Execution. Figure 18 shows a block diagram illustrating a Multi-Word Left Shift by 20 Instruction Execution. Figure 19 shows a block diagram illustrating a Multi-Word Right Shift by 4 Instruction Execution. Figure 20 shows a block diagram illustrating a Multi-Word Right Shift by 20 Instruction Execution. The multi word shift instructions rely on additional special registers.

32-bit left shifts

[0264] The multi-word left shift instructions utilize the shifter associated with the ACCn registers. The instruction can shift 0 to 31 positions. Although the shifter can only implement shifts of up to 15 positions to the left, by rearranging the storing into the destination registers an apparent shift of 31 positions may be obtained.

[0265] The Multi-Word Left Shift By 4 Instruction Execution (see Figure 17) provides an example where the shift amount is 15 or less. The Wnd destination register is aligned with the source and the CARRY0 register contains the shift out results. The CARRY1 register is unused and remains cleared. When the next 16-bit word is shifted, the results are OR'ed with the contents of the CARRY0 register, providing the shift in from the previous shift. The SLMK instruction may be repeated for each 16-bit segment of the multi-word shift.

[0266] The Multi-Word Left Shift By 20 Instruction Execution (see Figure 18) provides an example where the shift amount is 16 or more. Here, the Wnd destination register is aligned to the right of the source, CARRY0 is aligned with the source and the CARRY1 register contains the shift out results. When the next 16-bit word is shifted, the results are OR'ed with the contents of the CARRY1 and CARRY0 register, providing the shift in from the previous shift. The SLMK instruction may be repeated for each 16-bit segment of the multi-word shift.

[0267] Note the shifter is shifting (20-16), making the shift equivalent to the previous example. When the instruction detects a shift value greater than 15, it is only necessary to realign the result registers and perform a smaller shift.

32-bit RIGHT shifts

[0268] The multi-word right shift instructions are similar to the left shifts. The Multi-Word Right Shift By 4 Instruction Execution (see Figure 19) provides an example where the shift amount is 15 or less. The Wnd destination register is aligned with the source and the CARRY1 register contains the shift out results. The CARRY0 register is unused and remains cleared. When the next 16-bit word is shifted, the results are OR'ed with the contents of the CARRY1 register, providing the shift in from the previous shift. The SLMK instruction may be repeated for each 16-bit segment of the multi-word shift.

[0269] The Multi-Word Right Shift By 20 Instruction Execution (see Figure 20) provides an example where the shift amount is 16 or more. Here, the Wnd destination register is aligned to the left of the source, CARRY1 is aligned with the source and the CARRY0 register contains the shift out results. When the next 16-bit word is shifted, the results are OR'ed with the contents of the CARRY1 and CARRY0 register, providing the shift in from the previous shift. The SLMK instruction may be repeated for each 16-bit segment of the multi-word shift.

[0270] Note that the examples given show arithmetic shifts. If logical shifts are used, zeros would replace the sign bits.

16-Bit Shifts

[0271] The ASR, LSR and SL instructions allow for shifts of 16-bit words. The shift value should be limited to 15 positions by the user for useful results.

Multi-Word Shifts On Words Longer Than 32 Bits

[0272] The MSL and MSR instructions allow for shifts of words greater than 32 bits. This may be useful for IP addresses or encryption keys. Note that the shift is still limited up to 31 positions. For example, to shift a 64 bit word:

```

; W3...W0 - source word (ms...ls)
; W7...W4 - destination word (ms...ls)
; W8 - shift value (0..31)
Code:      LSR  W3,W8,W7
          MSR  W2,W8,W6
          MSR  W1,W8,W5
          MSR  W0,W8,W4

```

Multi-Word Rotates

[0273] Because the CARRY registers are readable, the multi-word shift instructions may be used for rotates. For example, to left rotate a 16 bit word:

```

; W1 - source word
; W0 - destination word (default Ww)
; W8 - rotate value (0..15)
Code:      SL   W1,W8,W0
          IOR  CARRY0,Ww

```

For example, to left rotate a 32 bit word:

```

; W1...W0 - source word (ms...ls)
; W3...W2 - destination word (ms...ls)
; W4 - rotate value (0..31)
; W5,W6 - temporaries
Code:  SL  W0,W4,W2
      MSL  W1,W4,W3
      MOV  CARRY0,W5
      MOV  CARRY1,W6
      IOR  W5,W2,W2  ;carry0+dest(ls)
      IOR  W6,W3,W3  ;carry1+dest(ms)

```

[0274] Using the MSL and MSR instructions, rotates of greater word lengths may be achieved.

DSP Data FormatsInteger and Fractional Data

[0275] The dsPIC DSP core supports integer and fractional data operations. Data format selection is made by the IF bit in the DSP control register CORCON<0>. Setting this bit to “1” selects integer mode; setting this bit to “0” selects fractional mode.

[0276] Integer data is inherently represented as a signed two’s-complement value, where the MSB is defined as a sign bit. Generally speaking, the range of an N-bit two’s complement integer is -2^{N-1} to $2^{N-1}-1$. For a 16-bit integer, the data range is –32768 (0x8000) to 32767 (0x7FFF), including 0 (see Figure 1). For a 32-bit integer, the data range is –2,147,483,648 (0x8000 0000) to 2,147,483,645 (0x7FFF FFFF).

[0277] When the dsPIC is in fractional mode, data is represented as a two’s complement fraction where the MSB is defined as a sign bit and the radix point is implied to lie just after the sign bit (Q1.X format). The range of an N-bit two’s complement fraction with this implied radix point is -1.0 to $(1-2^{1-N})$. For a 16-bit fraction, the Q1.15 data range is -1.0 (0x8000) to 0.999969482 (0x7FFF), including 0 (see Figure 1) and has a precision of 3.01518×10^{-5} . In fractional mode, the 16x16 dsPIC multiplier generates a Q1.31 product which has a precision of 4.65661×10^{-10} . Figure 21 shows a block diagram illustrating a 16-Bit integer and fractional modes.

Super Saturation Mode

[0278] The SATMOD bit, CORCON<3>, enables Super Saturation mode and expands the dynamic range of the accumulators by using 8 guard bits. When the SATMOD bit is set to “1”, Super Saturation mode is enabled and the 40-bit accumulators support an integer range of -5.498×10^{11} (0x80 0000 0000) to 5.498×10^{11} (0x7F FFFF FFFF).

In fractional mode, the guard bits of the accumulator do not modify the location of the radix point and the 40-bit accumulators use Q9.31 fractional format. Note that all fractional operation results are stored in the 40-bit accumulator justified with a Q1.31 radix point. As in integer mode, the guard bits merely increase the dynamic range of the accumulator. Q9.31 fractions have a range of -256.0 (0x80 0000 0000) to $(256.0 - 4.65661 \times 10^{-10})$ (0x7F FFFF FFFF). See Section 2.3.3 of the Core DOS for a description of the dsPIC overflow and saturation modes.

Scaling and Normalizing With FBCL Instruction

[0279] To minimize quantization errors that are associated with data processing using DSP instructions, it is important to utilize the complete available resolution of the dsPIC register set. This may require scaling data up to avoid underflows (i.e., when processing data from a 12-bit ADC) or scaling data down to avoid overflows (i.e., when sending data to a 10-bit DAC). The scaling which must be performed to minimize quantization errors depends on the dynamic range of the input data which is operated on, and the requirements of the dynamic range of the output data. At times these conditions may be known apriori and fixed scaling may be employed. Other times, scaling conditions may be not be fixed or known, and then dynamic scaling must be used to process data.

[0280] The Find First Bit Change Left (FBCL) instruction can effeciently be used to perform dynamic scaling. The FBCL function determines the exponent of the byte or word which it operates on (namely the amount which the value may be shifted before overflowing), and stores the exponent such that it may be used to later scale the value by shifting. The exponent is determined by detecting the first bit change starting from the sign bit and working towards the LSB. Scaling Examples shows data with various dynamic

ranges, their exponents, and the value after scaling each data to maximize the dynamic range.

TABLE 169 -- SCALING EXAMPLES

<u>Data Value</u>	<u>Exponent</u>	<u>Scaled Value for Max Dynamic Range</u> <u>(Data Value << Exponent)</u>
<u>0x0001</u>	<u>14</u>	<u>0x4000</u>
<u>0x0002</u>	<u>13</u>	<u>0x4000</u>
<u>0x0004</u>	<u>12</u>	<u>0x4000</u>
<u>0x0100</u>	<u>6</u>	<u>0x4000</u>
<u>0x0101</u>	<u>6</u>	<u>0x4040</u>
<u>0x01FF</u>	<u>6</u>	<u>0x7FC0</u>
<u>0x0806</u>	<u>3</u>	<u>0x4030</u>
<u>0x2007</u>	<u>1</u>	<u>0x400E</u>
<u>0x4800</u>	<u>0</u>	<u>0x4800</u>
<u>0x7000</u>	<u>0</u>	<u>0x7000</u>
<u>0x8000</u>	<u>0</u>	<u>0x8000</u>
<u>0x900A</u>	<u>0</u>	<u>0x900A</u>
<u>0xE001</u>	<u>2</u>	<u>0x8004</u>
<u>0xFF07</u>	<u>7</u>	<u>0x8380</u>
<u>0xFFFF</u>	<u>0</u>	<u>0xFFFF*</u>
<u>*A "hole" where FBCL fails to detect the correct exponent</u>		

[0281] As a practical example, assume that block processing is performed on a sequence of data with very low dynamic range stored in Q1.15 fractional format. To minimize quantization errors, the data may be scaled up to prevent any quantization loss which may occur as it is processed. The FBCL instruction can be executed on the sample with the largest magnitude to determine the optimal scaling value for processing the data. Note that scaling the data up is performed by left shifting the data (see Section 2.2 of the Core DOS for a description of the Barrel Shifter). This is demonstrated with the code snippet below.

```

; assume W0 contains the largest absolute value of the data
block

; assume W4 points to the beginning of the data block

; assume the block of data contains BLOCK SIZE words

; determine the exponent to use for scaling

FBCL W0, W2 ; store exponent in W2

```

; scale the entire data block by the optimal amount before processing

DO SCALE LOOP, BLOCK SIZE

MOV [W4], W1

; move the next data sample to W1

SLW W1, W2, W3

; shift W1 by W2 bits and store to W3

SCALE LOOP:

MOV W3, [W4]++

;store scaled input (overwrite original)

; now process the data

; (processing block goes here)

Accumulator Normalization With FBCL

[0282] The process of scaling a quantized value for its maximum dynamic range is known as normalization (the data in the third column in Table 169: Scaling Examples, contains normalized data). Accumulator normalization is a technique used to ensure that the accumulator is properly aligned before storing data from the accumulator, and the FBCL instruction facilitates this function.

[0283] The two 40-bit accumulators each have 8 guard bits which expand the accumulator from Q1.31 to Q9.31 when operating in Super Saturation mode. Even in

Super Saturation mode the Store Accumulator (SAC) instruction only stores 16-bit data (in Q1.15 format) from ACC<31:16>.

[0284] Proper data alignment for storing the contents of the accumulator may be achieved by scaling the accumulator down if the guard bits are in use, or scaling the accumulator up if all of the accumulator high bits are not being used. To perform such scaling, the FBCL instruction must operate on the guard bits in byte mode and it must operate on the high accumulator in word mode. If a shift is required, the ALU's 40-bit shifter is employed using the SFTAC instruction to perform the scaling. Listed below is a code snippet for accumulator normalization.

```

; assume an operation in ACCA has just completed (status bits
are intact)
; assume the processor is in super saturation mode
; assume W4 points to the ACCA guard byte (0x44)
; assume W5 points to the ACCA high word (0x42)
BOA FBCL GUARD ; if overflow we right shift
FBCL HI:
FBCL [W5], W0 ; extract exponent for left shift
BRA SHIFT ACC ; branch to the shift
FBCL GUARD:
FBCL.B [W4], W0 ; extract exponent for right shift
ADDLS.B W0, 8, W0 ; adjust the sign for right shift
SHIFT ACC:
SFTAC W0 ; shift the accumulator to normalize

```

The above code assumes that negative values are returned by FBCL to facilitate scaling up.

DO operations

[0285] The DO instructions implement simple looping. The instruction will execute a set of instructions a certain number of times. The loop count is selected with a constant or a W register. The loop will be executed n+1 times. For a W register, only the LS 14-bits are significant. The DO instruction loads the LSR register with the value of the PC after the DO instruction. It adds the loop offset to that PC and loads that value to the LER

register. It then continues to execute code starting with PC+2 until the PC matches the LER. When PC matches LER, the loop count is compared to negative. If not, the PC is loaded with the LSR value to branch back to the loop start. The loop count is decremented. When the loop count compares negative, the next sequential instruction executes. The instructions in the loop need not be consecutive. Figure 22 shows a block diagram illustrating a DO operation. Figure 23 shows a block diagram illustrating an alternate embodiment of the DO operation.

[0286] [0048] The instruction set coding is illustrated with reference to ~~Table 1~~ Tables 2 through 162 which ~~depicts~~ depict the PLA mnemonic for each instruction, its assembly syntax, a corresponding description and its corresponding 24 bit opcode. Each of these opcodes is unique and provides a basis for the instruction fetch/decode 110 to derive and transmit different control signals to each processor element to selectively involve that element in the instruction processing. Table 1-~~3~~ 188 sets forth status flag operations for the instruction set.

[0049] ~~Table 4 depicts opcode field descriptions for the designated instruction set which are referenced in Table 1-2.~~

[0287] [0050] The instruction set may be grouped into the following functional categories: move instructions; math instructions; rotate/shift instructions; bit instructions; DSP instructions; skip instructions; flow instructions and stack instructions.

[0288] [0051] Table 1-~~5~~ 190 depicts addressing modes for source registers. Table 1-~~6~~ 191 depicts addressing modes for destination registers. Table 1-~~7~~ 190 depicts offset addressing modes for WSO source registers. Table 1-~~8~~ 193 depicts offset addressing modes for WSO destination registers. Tables 1-~~9~~ 194 through 1-~~14~~ 199 depict examples of prefetch operations and MAC operations.

[0052] ~~The instruction field coding which breaks down the opcode into fields exploited by the instruction decoder is shown in Table 2-1. The opcodes are mapped to simplify the instruction~~

~~decoding logic.~~ [0053] Collectively, the Tables illustrate the composition of the instruction opcode, the mnemonics that are assigned to the opcodes and details of the operation of the instruction. ~~Even more details regarding each designated instruction and its exemplary uses according to an embodiment of the present invention are presented in Appendix A. Illustrative details regarding addressing modes are presented in Appendix B. An embodiment of timing for instructions within the instruction set is presented graphically in Appendix C. A detailed embodiment of an architecture for executing the instruction set is attached as Appendix D. The Appendices are incorporated by reference herein.~~

[0289] [0054] The following terms, used in the Appendices, are intended to specify an illustrative embodiment of a processor, such as a digital signal controller, that may be used to implement the instruction set according to the present invention: "RoadRunner" and "dsPIC." Other embodiments may be implemented as a matter of design choice.

[0055] Instruction Flows

Address Generator Units

[0290] The following description is enhanced by reference to Figures 24-81. Figure 24 shows a block diagram illustrating a Register Direct mode. Figure 25 shows a block diagram illustrating an alternate Register Indirect addressing mode. Figure 26 shows a block diagram illustrating a Register Indirect with Post-Decrement addressing mode. Figure 27 shows a block diagram illustrating a Register Indirect with Post-Increment addressing mode. Figure 28 shows a block diagram illustrating a Register Indirect with Pre-Decrement addressing mode. Figure 29 Register Indirect with Pre-Increment Addressing mode. Figure 30 shows a block diagram illustrating a Register Direct with 5-bit signed Literal Operation mode.

[0291] Figure 31 shows a block diagram illustrating a Register Direct, Operand Source mode. Figure 32 shows a block diagram illustrating a Register Indirect, Result Destination mode. Figure 33 shows a block diagram illustrating a Register Indirect, Operand Source mode. Figure 34 shows a block diagram illustrating a Register Indirect,

Result Destination mode. Figure 35 shows a block diagram illustrating a Register Indirect with Post Decrement, Source Operand mode. Figure 36 shows a block diagram illustrating a Register Indirect with Post Decrement, Result Destination mode. Figure 37 shows a block diagram illustrating a Register Indirect with Post Increment, Operand Source mode. Figure 38 shows a block diagram illustrating a Register Indirect with Post Increment, Result Destination mode.

[0292] Figure 39 shows a block diagram illustrating a Register Indirect with Pre-Decrement, Source Operand mode. Figure 40 shows a block diagram illustrating a Register Indirect with Pre-Decrement, Result Destination mode. Figure 41 shows a block diagram illustrating a Register Indirect with Pre-Increment, Source Operand mode. Figure 42 shows a block diagram illustrating a Register Indirect with Pre-Increment, Result Destination mode. Figure 43 shows a block diagram illustrating a Register Direct, Operand Source mode. Figure 44 shows a block diagram illustrating a Register Direct, Result Destination mode. Figure 45 shows a block diagram illustrating a Register Indirect, Source Operand mode.

[0293] Figure 46 shows a block diagram illustrating a Register Indirect, Result Destination mode. Figure 47 shows a block diagram illustrating a Register Indirect with Post Decrement, Source Operand mode. Figure 48 shows a block diagram illustrating a Register Indirect with Post Decrement, Result Destination mode. Figure 49 shows a block diagram illustrating a Register Indirect with Post Increment, Source Operand mode. Figure 50 shows a block diagram illustrating a Register Indirect with Post Increment, Result Destination mode. Figure 51 shows a block diagram illustrating a Register Indirect with Pre-Decrement, Source Operand mode.

[0294] Figure 52 shows a block diagram illustrating a Register Indirect with Pre-Decrement, Result Destination mode. Figure 53 shows a block diagram illustrating a Register Indirect with Register Offset, Operand Source mode. Figure 54 shows a block diagram illustrating a Register Indirect with Register Offset, Result Destination mode. Figure 55 shows a block diagram illustrating a Register Indirect with Constant Offset, Source Operand mode. Figure 56 shows a block diagram illustrating a Register Indirect with Constant Offset, Result Destination mode. Figure 57 shows a block diagram illustrating a Register Indirect with Pre-Decrement, Source Operand mode. Figure 58 shows a block diagram illustrating a Register Indirect with Pre-Decrement, Result Destination mode.

[0295] Figure 59 shows a block diagram illustrating a Register Indirect mode. Figure 60 shows a block diagram illustrating a Register Indirect with Post Increment mode. Figure 61 shows a block diagram illustrating a Register Indirect with Register Offset Operand Source mode. Figure 62 shows a block diagram illustrating a Register Indirect with Post Decrement mode.

[0296] Figure 63 shows a block diagram illustrating an X AGU. Figure 64 shows a block diagram illustrating a Y AGU. Figure 65 shows a block diagram illustrating an Incrementing Buffer Modulo addressing operation. Figure 66 shows a block diagram illustrating a Decrementing Buffer Modulo addressing operation. Figure 67 shows a block diagram illustrating a Bit Reversed EA calculation. Figure 68 shows a block diagram illustrating a Alternative Bit Reversed EA calculation method.

[0297] Figure 69 shows a block diagram illustrating a Bit Reversed Addressing, Source Operand mode. Figure 70 shows a block diagram illustrating a Bit Reversed

Addressing, Destination Operand mode. Figure 71 shows a block diagram illustrating a Register Indirect, Table Read Operand Destination mode. Figure 72 shows a block diagram illustrating a Register Indirect, Table Read Operand Source mode. Figure 73 shows a block diagram illustrating a Register Indirect, Table Read Result Destination mode. Figure 74 shows a block diagram illustrating a Register Indirect with Post Decrement, Table Read Source Operand mode. Figure 75 shows a block diagram illustrating a Register Indirect with Post Decrement, Table Read Result Destination mode. Figure 76 shows a block diagram illustrating a Register Indirect with Post Increment, Table Read Operand Source mode. Figure 77 shows a block diagram illustrating a Register Indirect with Post Increment, Table Read Result Destination mode. Figure 78 shows a block diagram illustrating a Register Indirect with Pre-Decrement, Table Read Source Operand mode. Figure 79 shows a block diagram illustrating a Register Indirect with Pre-Decrement, Table Read Result Destination mode. Figure 80 shows a block diagram illustrating a Register Indirect with Pre-Increment, Table Read Source Operand mode. Figure 81 shows a block diagram illustrating a Register Indirect with Pre-Increment, Table Read Result Destination.

[0298] The dsPIC core contains two independent address generator units. The X AGU is for MCU and DSP instructions. The Y AGU is for DSP MAC class of instructions only. They are capable of supporting three types of data addressing:

- Linear addressing
- Modulo (circular) addressing
- Bit Reversed addressing (X AGU only)

[0299] Linear and modulo data addressing modes can be applies to data space or program space. Although bit reversed addressing will work with any EA calculation, by definition it is only applicable to data space.

Data Space Organization

[0300] Although the data space memory is organized as 16-bit words, all effective addresses (EAs) point to bytes. Instructions can thus access any byte or aligned words (data words at an even address). Misaligned word accesses are not supported, and if attempted will initiate an address error trap. The LS-bit of the EA is used to determine upper or lower byte access. The LS-bit becomes a 'don't care' for word accesses. Each memory (or register where appropriate) must provide independent upper and lower byte write lines to support byte writes. In addition, a multiplexor must be included to route the LS byte of an operand to the upper or lower byte of the target EA word for both reads and writes.

[0301] When executing instructions which require just one source operand to be fetched from data space, the X AGU is used to calculate the effective address. The AGU can generate an address to point to anywhere in the 64K byte data space. It supports all addressing modes, modulo addressing for low overhead circular buffers, and bit reversed addressing to facilitate FFT data reorganization.

[0302] When executing instructions which require two source operands to be concurrently fetched (i.e. the MAC class of DSP instructions), both the X and Y AGUs are used simultaneously and the data space is split into two independent address spaces, X and Y. The Y AGU supports register indirect post-modified and modulo addressing only. Note that the data write phase of the MAC class of instruction does not split X and Y address

space. The write EA is calculated using the X AGU and the data space is configured for full 64Kbyte access.

[0303] In the split data space mode, some W register address pointers are dedicated to AGU X, others to AGU Y (see Figures 63 and 64, respectively). The EAs of each operand must therefore be restricted to be within different address spaces. If they are not, one of the EAs will be outside the address space of the corresponding data space (and will fetch the bus default value, 0x0000).

Instruction Addressing Modes

[0304] While alternate addressing modes are possible with the present invention, the basic set of addressing modes for this illustrative example are shown in Table 170. Note that, 'Wn+=' indicates that the contents of Wn is added to something to form the effective address which is then written back into Wn. 'Wn+' indicates that the contents of Wn is added to something to form the effective address but the contents of Wn remain unchanged.

[0305] The addressing modes in Table 170 form the basis of three groups of addressing modes optimized to support specific instruction features. They are Mode 1, Mode 2 and Mode 3. The DSP MAC and derivative instructions are an exception where the addressing modes are encoded differently. This set of addressing modes is referred to as Mode 4. Refer to dsPIC Instruction Set DOS for full details.

TABLE 170 -- Fundamental Addressing Modes Supported

<u>Addressing Mode</u>	<u>Function</u>	<u>Description</u>
<u>Register Direct</u>	<u>EA = Wn</u>	<u>Wn is the EA</u>
<u>Register Indirect</u>	<u>EA = [Wn]</u>	<u>The contents of Wn forms the EA</u>
<u>Register Indirect Post-modified</u>	<u>EA = [Wn] += 1</u> <u>EA = [Wn] -= 1</u>	<u>The contents of Wn forms the EA which is post-modified by a constant value</u>
<u>Register Indirect Pre-modified</u>	<u>EA = [Wn + 1]</u> <u>EA = [Wn - 1]</u>	<u>Wn is pre-modified by a signed constant value to form the EA</u>
<u>Register Indirect with</u>	<u>EA = [Wn + Wb]</u>	<u>The sum of Wn and Wb forms the EA</u>

Register OffsetRegister Indirect with
Constant Offset $EA = [Wn +$
constant]The sum of Wn and a signed constant value forms
the EAEA = effective addressAll address modification values (except Wb) are scaled for word access[0306] All but a few instructions support both 8-bit and 16-bit operand data sizes.In order to efficiently accommodate this requirement, all effective addresses are byte aligned. As the data space is 16-bits wide, the following consequences must be understood.

1. Miss-aligned word accesses are not supported. All word effective addresses must be even (the LS-bit of the EA is ignored by the data space memory).
2. The LS-bit of the effective address is used to select which byte (upper or lower) is multiplexed onto bits [7:0] of the data bus for byte sized accesses.
3. Post and pre-modification of a register by a constant value to create a new effective address must take into account of the data size accessed. All constant values, whether implied (e.g. post-inc) or declared (e.g. post-modify with $S5lit$) are scaled by a factor of 2 for word accesses. For example:

 $[Ws] += 1$ will post-modify data source pointer Ws by 1 for a byte access, and by 2 for a word access. $[Ws] += Slit5$ will post-modify data source pointer Ws by $Slit5$ for byte accesses and $Slit5 << 1$ (shift left by 1) for word accesses. Finally, register offsets are not scaled.[0307] Unless otherwise noted, it is assumed that all addresses and addressing modes refer to byte size accesses. All addressing modes which have to calculate the EA (pre-modified, register offset and constant offset) have very tight timing requirements which may require some instruction addressing sequence restrictions.Mode 1[0308] Mode 1 determines the addressing mode for one of the two operand sources required for the three operand instructions (found in categories 'MATH' and 'SKIP'). These instructions are of the form:

Result = Operand 1 <function> Operand 2

[0309] Operand1 is always a register (i.e. the addressing mode can only be register direct) which is referred to as Wb. Operand 2 is fetched from data memory based upon the addressing mode selected by Mode 1. Mode 1 therefore defines one of the source operand addressing modes and implies that of the other source operand.

[0310] In addition, Mode 1 may also provide a signed 5-bit constant (literal) as the operand. In this case, the instruction is of the form:

Result = Operand 1 <function> signed literal

[0311] Operand 1 is always a register (i.e. the addressing mode can only be register direct) which is selected from the Ws field in the instruction. The 4-bit Wb field forms the 4 LS-bits of a signed constant. It is concatenated with the LS-bit of the three bit Mode 1 field to form the 5-bit signed constant value.

[0312] In summary, Mode 1 supports the addressing modes shown in Table 171

TABLE 171 -- Mode 1 Addressing Mode Definition

<u>Mode 1</u>	<u>Operand 1</u>		<u>Operand 2</u>	
<u>Bit</u> <u>Encoding</u>	<u>Function</u>	<u>Description</u>	<u>Function</u>	<u>Description</u>
<u>000</u>	<u>EA = Wb</u>	<u>Register direct</u>	<u>EA = Ws</u>	<u>Register direct</u>
<u>001</u>	<u>EA = Wb</u>	<u>Register direct</u>	<u>EA = [Ws]</u>	<u>Register indirect</u>
<u>010</u>	<u>EA = Wb</u>	<u>Register direct</u>	<u>EA = [Ws]-= 1</u>	<u>Register indirect post-decremented</u>
<u>011</u>	<u>EA = Wb</u>	<u>Register direct</u>	<u>EA = [Ws]+= 1</u>	<u>Register indirect post-incremented</u>
<u>100</u>	<u>EA = Wb</u>	<u>Register direct</u>	<u>EA = [Ws-=1]</u>	<u>Register indirect pre-decremented</u>
<u>101</u>	<u>EA = Wb</u>	<u>Register direct</u>	<u>EA = [Ws+=1]</u>	<u>Register indirect pre-incremented</u>
<u>110</u>	<u>EA = Ws</u>	<u>Register direct</u>	<u>Operand 2 = S5lit</u>	<u>5-bit signed literal</u>
<u>111</u>				

Mode 1, Register Direct

[0313] Addressing Mode 1, Submode 0 is register direct. The implied effective address is the memory mapped address of register Ws. Rather than executing a memory

fetch, it may be preferable to perform two W-array fetches if bussing allows. The operand is contained in Ws as shown in Figure 24.

Mode 1, Register Indirect

[0314] Addressing Mode 1, Submode 1 is register indirect. The effective address contained in register Ws points to the operand as shown in Figure 25.

Mode 1, Register Indirect with Post Decrement

[0315] Addressing Mode 1, Submode 2 is register indirect with post decrement. The effective address contained in register Ws points to the operand. Ws is then post decremented as shown in Figure 26.

Mode 1, Register Indirect with Post Increment

[0316] Addressing Mode 1, Submode 3 is register indirect with post increment. The effective address contained in register Ws points to the operand. Ws is then incremented as shown in Figure 27.

Mode 1, Register Indirect with Pre Decrement

[0317] Addressing Mode 1, Submode 4 is register indirect with pre-decrement. Register Ws is decremented to form the effective address which points to the operand as shown in Figure 28.

Mode 1, Register Indirect with Pre Increment

[0318] Addressing Mode 1, Submode 5 is register indirect with pre increment. Register Ws is incremented to form the effective address which points to the operand as shown in Figure 29.

Mode 1, Register Direct with 5-bit Signed Literal

[0319] Addressing Mode 1, Submode 6/7 is register direct with 5-bit signed literal. As shown in Figure 30, operand 1 is contained in Ws. Operand 2 is the 5-bit signed literal embedded within the instruction. The 4-bit Wb field forms the 4 LS-bits of a signed constant. It is concatenated with the LS-bit of the three bit Mode 1 field to form the 5-bit signed constant value.

Mode 2

[0320] Mode 2 determines the addressing mode for either the result destination or a source operand, depending upon instruction requirements. It follows the same definition for each encoding as Mode 1 except that it applies to only one operand. The Mode 1 signed 5-bit constant value mode makes little sense where Mode 2 is used, and is therefore not supported. In summary, Mode 2 supports the addressing mode shown in Table 172.

TABLE 172 -- Mode 2 Addressing Mode Definition

<u>Mode 2 Bit Encoding</u>	<u>Function (Source)</u>	<u>Function (Destination)</u>	<u>Description</u>
<u>000</u>	<u>EA = Wsrc</u>	<u>EA = Wdst</u>	<u>Register direct</u>
<u>001</u>	<u>EA = [Wsrc]</u>	<u>EA = [Wdst]</u>	<u>Register indirect</u>
<u>010</u>	<u>EA = [Wsrc]-= 1</u>	<u>EA = [Wdst]-= 1</u>	<u>Register indirect post-decremented</u>
<u>011</u>	<u>EA = [Wsrc]+= 1</u>	<u>EA = [Wdst]+= 1</u>	<u>Register indirect post-incremented</u>
<u>100</u>	<u>EA = [Wsrc-=1]</u>	<u>EA = [Wdst-=1]</u>	<u>Register indirect pre-decremented</u>
<u>101</u>	<u>EA = [Wsrc+=1]</u>	<u>EA = [Wdst+=1]</u>	<u>Register indirect pre-incremented</u>
<u>110</u>	<u>Unused</u>	<u>Unused</u>	
<u>111</u>	<u>Unused</u>	<u>Unused</u>	

Mode 2, Register Direct

[0321] Addressing Mode 2, Submode 0 is register direct. The implied effective address is the memory mapped address of register Wsrc or Wdst. The operand is contained in Wsrc as shown in Figure 31, or the result is written to Wdst as shown in Figure 32. In both cases, Wsrc or Wdst is accessed through addressing its memory mapped image. Note that, as the EA is implicitly defined as a word address, byte data size accesses

will only be able to read or write the LS byte<7:0> (LS-bit of the EA is always clear) in this addressing mode. Rather than executing a memory fetch, it may be preferable to perform two W-array fetches if bussing allows???

Mode 2, Register Indirect

[0322] Addressing Mode 2, Submode 1 is register indirect. The effective address contained in register Wsrc points to the operand as shown in Figure 33, or Wdst points to the result destination as shown in Figure 34.

Mode 2, Register Indirect with Post Decrement

[0323] Addressing Mode 2, Submode 2 is register indirect with post decrement. The effective address contained in register Wsrc points to the operand, or the effective address contained in register Wdst points to the result destination. Wsrc or Wdst is then post decremented as shown in Figure 35 and Figure 36.

Mode 2, Register Indirect with Post Decrement

[0324] Addressing Mode 2, Submode 3 is register indirect with post decrement. The effective address contained in register Wsrc points to the source operand, or the effective address contained in register Wdst points to the result destination. Wsrc or Wdst are then decremented as shown in Figure 37 and Figure 38.

Mode 2, Register Indirect with Pre Decrement

[0325] Addressing Mode 2, Submode 4 is register indirect with pre decrement. Register Wsrc or Wdst is decremented to form the effective address which points to the operand as shown in Figure 39 and Figure 40

Mode 2, Register Indirect with Pre Increment

[0326] Addressing Mode 2, Submode 5 is register indirect with pre increment. Register Wsrc or Wdst is incremented to form the effective address which points to the operand as shown in Figure 41 and Figure 42.

Mode 3

[0327] Mode 3 is used by 'MOVE' and some of the DSP class instructions where addressing flexibility is important. It follows the same definition for each encoding as Mode 1 except that it uses the Wb field as an address operand (instead of a data operand). In addition, Mode 3 also supports register with register offset addressing mode, sometimes referred to as register indexed.

[0328] The 5-bit signed constant required by Submode 6/7 is created by concatenating the Wb field with the LS-bit of the 3-bit Mode 3 field. For the MOV instruction, the Mode 3 addressing modes can differ for the source and destination EA. However, the 4-bit Wb field is shared between both source and destination (but typically only used by one). In summary, Mode 3 supports the addressing mode shown in Table 173.

TABLE 173 -- Mode 3 Addressing mode Definition

<u>Mode 3 Bit Encoding</u>	<u>Function</u>	<u>Description</u>
<u>000</u>	<u>EA = Wn</u>	<u>Register direct</u>
<u>001</u>	<u>EA = [Wn]</u>	<u>Register indirect</u>
<u>010</u>	<u>EA = [Wdst]-= 1</u>	<u>Register indirect post-decremented</u>
<u>011</u>	<u>EA = [Wdst]+= 1</u>	<u>Register indirect post-incremented</u>
<u>100</u>	<u>EA = [Wdst= 1]</u>	<u>Register indirect pre-decrement</u>
<u>101</u>	<u>EA = [Wn + Wb]</u>	<u>Register indirect with register offset</u>
<u>110</u>	<u>EA = [Wn + S5lit]</u>	<u>Register indirect with signed 5-bit constant value</u>
<u>111</u>		<u>offset (note 1)</u>

Mode 3, Register Direct

[0329] Addressing Mode 3, Submode 0 is register direct. The implied effective address is the memory mapped address of register Wsrc or Wdst. The operand is

contained in Wsrc as shown in Figure 43, or the result is written to Wdst as shown in Figure 44. In both cases, Wsrc or Wdst is accessed through addressing its memory mapped image. Rather than executing a memory fetch, it may be preferable to perform two W-array fetches if bussing allows.

Mode 3, Register Indirect

[0330] Addressing Mode 3, Submode 1 is register indirect. The effective address contained in register Wsrc points to the operand as shown in Figure 45, or Wdst points to the result destination as shown in Figure 46.

Mode 3, Register Indirect with Post Decrement

[0331] Addressing Mode 3, Submode 2 is register indirect with post decrement. The effective address contained in register Wsrc points to the operand, or the effective address contained in register Wdst points to the result destination. Wsrc or Wdst is then post decremented as shown in Figure 47 and Figure 48.

Mode 3, Register Indirect with Post Modification

[0332] Addressing Mode 3, Submode 3 is register indirect with post-increment. The effective address contained in register Wsrc points to the operand or the effective address contained in register Wdst points to the result destination. Wsrc or Wdst are then incremented as shown in Figure 49 and Figure 50.

Mode 3, Register Indirect with Pre Decrement

[0333] Addressing Mode 2, Submode 4 is register indirect with pre decrement. Register Wsrc or Wdst is decremented to form the effective address which points to the operand as shown in Figure 51 and Figure 52.

Mode 3, Register Indirect with Register Offset

[0334] Addressing Mode 3, Submode 5 is register indirect with register offset. For an operand read, the effective address of the operand is formed by adding the contents of Wsrc and Wb as shown in Figure 53. For a result destination write, the effective address of the operand is formed by adding the contents of Wdst and Wb as shown in Figure 54. Wb, Wsc or Wdst are not modified by these operations unless bit reversed addressing (described elsewhere in this specification) is enabled, in which case Wsc and/or Wdst are updated with the new EA. This is the only addressing mode which operates in a meaningful way with bit reversed addressing.

Mode 3, Register Indirect with Constant Offset

[0335] Addressing Mode 3, Submode 6/7 is register indirect with constant offset. For an operand read, the effective address of the operand is formed by adding the contents of Wsrc and a 5-bit signed literal, as shown in Figure 55. For a result destination write, the effective address of the operand is formed by adding the contents of Wdst and a 5-bit signed literal as shown in Figure 56. Wsc or Wdst are not modified by these operations. The 4-bit Wb field forms the 4 LS-bits of the signed constant. It is concatenated with the LS-bit of the three bit Mode 1 field to form a 5-bit signed constant value. If the 5-bit signed literal equals 0, this addressing mode is interpreted as register indirect with a pre-decrement..

Mode 4

[0336] The dual source operand DSP instructions (MAC, CLRAC, MPYAC & MOVAC) utilize a simplified set of addressing modes (Mode 4) to allow the user to effectively manipulate the data pointers through register indirect tables.

[0337] Wsrc must be a member of the set {W4, W5, W6, W7}. For data reads, W4 and W5 will always be directed to the X AGU and W6 and W7 will always be directed to the Y AGU. The effective addresses generated (before and after modification) must therefore be valid addresses within X data space for W4 and W5, and Y data space for W6 and W7. Register indirect with register offset addressing is only available for W5 (in X space) and W7 (in Y space).

[0338] In summary, Mode 4 supports the addressing modes shown in Table 174 for X data space and those shown in Table 175 for Y data space.

TABLE 174 -- Mode 4 Addressing Mode Definition for X Data Space

<u>Mode 4 Bit Encoding</u>	<u>Function</u>	<u>Description</u>
<u>0000</u>	<u>EA = [W 4]</u>	<u>Register indirect</u>
<u>0001</u>	<u>EA = [W 4] += 2</u>	<u>Register indirect post-inc by 2</u>
<u>0010</u>	<u>EA = [W 4] += 4</u>	<u>Register indirect post-inc by 4</u>
<u>0011</u>	<u>EA = [W 4] += 6</u>	<u>Register indirect post-inc by 6</u>
<u>0100</u>	<u>None</u>	<u>Disable data pre-fetch</u>
<u>0101</u>	<u>EA = [W 4] -= 6</u>	<u>Register indirect post-dec by 6</u>
<u>0110</u>	<u>EA = [W 4] -= 4</u>	<u>Register indirect post-dec by 4</u>
<u>0111</u>	<u>EA = [W 4] -= 2</u>	<u>Register indirect post-dec by 2</u>
<u>1000</u>	<u>EA = [W 5]</u>	<u>Register indirect</u>
<u>1001</u>	<u>EA = [W 5] += 2</u>	<u>Register indirect post-inc by 2</u>
<u>1010</u>	<u>EA = [W 5] += 4</u>	<u>Register indirect post-inc by 4</u>
<u>1011</u>	<u>EA = [W 5] += 6</u>	<u>Register indirect post-inc by 6</u>
<u>1100</u>	<u>EA = [W 5 + W 8]</u>	<u>Register indirect with register offset (indexed)</u>
<u>1101</u>	<u>EA = [W 5] -= 6</u>	<u>Register indirect post-dec by 6</u>
<u>1110</u>	<u>EA = [W 5] -= 4</u>	<u>Register indirect post-dec by 4</u>
<u>1111</u>	<u>EA = [W 5] -= 2</u>	<u>Register indirect post-dec by 2</u>

Mode 4 instructions are word sized only, so post-modification values are already scaled appropriately

Addressing mode defined by read address space

TABLE 175 -- Mode 4 Addressing mode Definition for Y Data Space

<u>Mode 4 Bit Encoding</u>	<u>Function</u>	<u>Description</u>
<u>0000</u>	<u>EA = [W 6]</u>	<u>Register indirect</u>
<u>0001</u>	<u>EA = [W 6] += 2</u>	<u>Register indirect post-inc by 2</u>

<u>0010</u>	<u>EA = [W 6]+=4</u>	<u>Register indirect post-inc by 4</u>
<u>0011</u>	<u>EA = [W 6]+=6</u>	<u>Register indirect post-inc by 6</u>
<u>0100</u>	<u>None</u>	<u>Disable data pre-fetch</u>
<u>0101</u>	<u>EA = [W 6]-=6</u>	<u>Register indirect post-dec by 6</u>
<u>0110</u>	<u>EA = [W 6]-=4</u>	<u>Register indirect post-dec by 4</u>
<u>0111</u>	<u>EA = [W 6]-=2</u>	<u>Register indirect post-dec by 2</u>
<u>1000</u>	<u>EA = [W 7]</u>	<u>Register indirect</u>
<u>1001</u>	<u>EA = [W 7]+=2</u>	<u>Register indirect post-inc by 2</u>
<u>1010</u>	<u>EA = [W 7]+=4</u>	<u>Register indirect post-inc by 4</u>
<u>1011</u>	<u>EA = [W 7]+=6</u>	<u>Register indirect post-inc by 6</u>
<u>1100</u>	<u>EA = [W 7 +W 8]</u>	<u>Register indirect with register offset</u>
<u>1101</u>	<u>EA = [W 7]-=6</u>	<u>Register indirect post-dec by 6</u>
<u>1110</u>	<u>EA = [W 7]-=4</u>	<u>Register indirect post-dec by 4</u>
<u>1111</u>	<u>EA = [W 7]-=2</u>	<u>Register indirect post-dec by 2</u>

Mode 4 instructions are word sized only, so post-modification values are already scaled appropriately

Addressing mode defined by read address space

Mode 4, Register Indirect

[0339] Addressing Mode 4, Submodes 0 & 8 are register indirect. The effective address contained in register Wsrc points to the operand as shown in Figure 59. Only word sized operands are allowed.

Mode 4, Register Indirect with Post Increment

[0340] Addressing Mode 4, Submodes 1, 2, 3, 9, 10 & 11 are register indirect with post increment. The effective address contained in register Wsrc points to the operand. Wsrc is then post incremented by 2, 4 or 6 as shown in Figure 60. Misaligned word fetches are possible if Wsrc contains an odd value. Should this occur, an address error trap will be generated.

Mode 4, Pre-fetch Inhibit

[0341] Addressing mode Mode 4, Submode 4 will inhibit a data fetch from X or Y address space. No target registers are modified.

Mode 4, Register Indirect with Register Offset

[0342] Addressing Mode 4, Submodes 12 is register indirect with register offset. The effective address of the operand is formed by adding the contents of Wsrc (W5 or W7) and W8 as shown in Figure 61. The offset register is fixed as W8. Neither Wsrc or W8 are not modified by these operations. This addressing mode operates in an identical manner to that of Mode 3 register indirect with register offset, in which the offset register (W8 in this case) is not automatically scaled for word accesses. Consequently, misaligned word fetches are possible if W8 contains an odd value. Should this occur, an address error trap will be generated.

[0343] Addressing Mode 4, Submodes 5, 6, 7, 13, 14 & 15 are register indirect with post decrement. The effective address contained in register Wsrc points to the operand. Wsrc is then post decremented by 2, 4 or 6 as shown in Figure 62. Misaligned word fetches are possible if Wsrc contains an odd value. Should this occur, an address error trap will be generated.

X AGU

[0344] The X AGU supports all addressing modes including modulo addressing and bit reversed addressing. A block diagram is shown in Figure 63. The basic elements are now described.

Effective Address Adder

[0345] The effective address (EA) adder generates the effective addresses for all instruction using X data space prior to modification by modulo addressing. It supports all addressing modes including bit reversed addressing. The adder accepts the source or destination W register on the A input and either of the following on B input based upon which addressing mode is required.

1. Offset (Wb) register contents
2. Signed 5-bit literal, Slit5
3. Constant value of: 0, +1, +2, +4, +6, -1, -2, -4 or -6

The value range for Slit5 is $-16 \leq \text{Slit5} \leq +15$.

Modulo and Bit Reversed Addressing Controller

[0346] The Modulo and Bit Reversed Addressing Controller block enables or disables these addressing modes, and provides the appropriate control signals to the rest of the AGU. If modulo and bit reversed addressing are disabled, the EA adder result passes unmodified to the AGU output.

Modulo Addressing Comparator/Subtractor

[0347] Modulo addressing relies on automatic correction of any generated EA such that it is forced back into the selected circular buffer address range. For an incrementing buffer, the offset sign is positive. The end address is therefore routed to the subtractor, and subtracted from the new EA. If the result is negative, the address is within the buffer boundaries and will propagate unchanged. If the result is positive (including zero), indicating the EA has passed the end address, it is logically ORed with the start address. This is equivalent to adding it to the start address to create the wrap address for a start address on a 'zero' power of two boundary.

[0348] For a decrementing buffer, the offset sign is negative. The start address is therefore routed to the subtractor, and subtracted from the new EA. If the result is positive, the address is within the buffer boundaries and will propagate unchanged. If the result is negative, indicating the EA has passed the start address, it is logically AND'ed with the start address. This is equivalent to adding it (a negative value) to the start address to create the wrap address for an end address on a 'ones' address boundary.

Y AGU

[0349] As the Y AGU is only used by the MAC class of DSP instructions, its function is restricted to supporting post-modified register indirect (using a constant modifier) and modulo addressing. A block diagram is shown in Figure 64. The basic elements are now described.

Effective Address Adder

[0350] The effective address (EA) Adder generates the effective addresses for all instruction using Y data space prior to modification by modulo addressing. It supports post-modified register indirect (using a constant modifier). It does not support bit reversed addressing. The adder accepts the source or destination W register on the A input and a constant (0, +2, +4, +6, -2, -4 or -6) on B input, depending upon the post modified constant declared in the instruction.

Modulo Addressing Controller

[0351] The Modulo Addressing Controller block enables or disables modulo addressing, and provides the appropriate control signals to the rest of the AGU. If modulo addressing is disabled, the EA adder result passes unmodified to the AGU output.

Modulo Addressing Comparator/Subtractor

[0352] Modulo addressing relies on automatic correction of any generated EA such that it is forced back into the selected circular buffer address range. For an incrementing buffer, the offset sign is positive. The end address is therefore routed to the subtractor, and subtracted from the new EA. If the result is negative, the address is within the buffer boundaries and will propagate unchanged. If the result is positive (including zero), indicating the EA has passed the end address, it is logically ORed with the start address.

This is equivalent to adding it to the start address to create the wrap address for a start address on a 'zero' power of two boundary.

[0353] For a decrementing buffer, the offset sign is negative. The start address is therefore routed to the subtractor, and subtracted from the new EA. If the result is positive, the address is within the buffer boundaries and will propagate unchanged. If the result is negative, indicating the EA has passed the start address, it is logically AND'ed with the start address. This is equivalent to adding it (a negative value) to the start address to create the wrap address for an end address on a 'ones' address boundary.

Modulo Addressing

[0354] Modulo addressing is a method of providing an automated means to support circular data buffers using hardware. The objective is to remove the need for software to perform data address boundary checks when executing tightly looped code as is typical in many DSP algorithms.

[0355] dsPIC modulo addressing can operate in either data or program space (since the data pointer mechanism is essentially the same for both). One circular buffer can be supported in each of the X (which also provides the pointers into Program space) and Y data spaces. Modulo addressing can operate on any W register pointer.

[0356] In order to minimize the hardware size for modulo addressing support, certain usage restrictions may be imposed. In summary, any one circular buffer can only be allowed to operate in one direction as the buffer start address (for incrementing buffers) or end address (for decrementing buffers) is restricted based upon the direction of the buffer. The direction is determined from the address offset sign.

Start and End Address

[0357] The modulo addressing scheme requires that either a starting or an end address be specified and loaded into the 16-bit modulo buffer address registers, XMODSRT, XModEND, YMODSRT, YModEND.

[0358] The data buffer start address is arbitrary but must be at a 'zero', power of two boundary for incrementing address buffers. It can be any address for decrementing address buffers. For example, if the buffer size (modulus value) is chosen to be 100 bytes (0x64), then the buffer start address for an incrementing buffer must contain 7 least significant zeros. Valid start addresses may therefore be 0xXX00 and 0xXX80 where 'x' is any hexadecimal value. Adding the buffer length to this value will give the end address to be written into X/YModEND. For example, if the start address was chosen to be 0x2000, then the X/YModEND would be set to $(0x2000 + 0x0064) = 0x2064$. Note that the last physical address of the buffer will be at end address -1 because the buffer range is 0 to 0x63. 'Starting address' refers to the smallest address boundary of the circular buffer. The initial entry address (first access of the buffer) may point to any address within the modulus range.

[0359] The data buffer end address is arbitrary but must be at a 'ones' boundary for decrementing buffers. It can be at any address for an incrementing buffer. For example, if the buffer size (modulus value) is chosen to be 100 bytes (0x64), then the buffer end address for an incrementing buffer must contain 7 least significant ones. Valid end addresses may therefore be 0xFF and 0x7F where 'X' is any hexadecimal value. Subtracting the buffer length from this value then adding 1 will give the start address to be written into X/YMODSRT. For example, if the end address was chosen to be 0x207F, then

the start address would be $(0x207F - 0x0064 + 1) = 0x201C$, which is the first physical address of the buffer.

[0360] In an incrementing buffer, the modulo addressing hardware performs the address correction by subtracting the buffer end address from the EA and, if the result is positive, adding it to the start address. As the start address is on a 'zero', power of two boundary, the addition may be performed by a logical OR operation.

[0361] In a decrementing buffer, the modulo addressing hardware performs the address correction by subtracting the buffer start address from the EA and, if the result is negative, adding it to the end address. As the end address is on a 'ones' boundary, the addition may be performed by a logical AND operation. All modulo addressing EA calculations assume word size data (LS-bit of every EA is always clear). The XM value may scaled accordingly to generate compatible (byte) addresses, leaving the LS-bit of all EAs clear.

Buffer Length

[0362] The data buffer length can be any value up to 64K words. The buffer length is not used in this scheme to correct buffer addresses or determine modulo range.

W Address Register Selection

[0363] The modulo and bit reversed addressing control register MODCON<15:0> contains enable flags plus W register field to specify the W address registers. The XWM and YWM fields selects which registers will operate with modulo addressing. If XWM = 15, AGU X modulo addressing is disabled. Similarly, if YWM = 15, AGU Y modulo addressing is disabled.

[0364] Modulo addressing and bit reversed addressing should not be enabled together. In the event that the user attempts to do this, bit reversed addressing will assume priority when active and X modulo addressing will be disabled.

[0365] The X address space pointer W register (XWM) to which modulo addressing is to be applied, is stored in MODCON<3:0> (see Table 176). Modulo addressing is enabled for X data space when XWM is set to any value other than 15 and the XModeN bit is set at MODCON[15].

[0366] The Y address space pointer W register (YWM) to which modulo addressing is to be applied, is stored in MODCON<7:4> (see Table 177). Modulo addressing is enabled for Y data space when YWM is set to any value other than 15 and the YModeN bit is set at MODCON[14].

Modulo Addressing Applicability

[0367] Modulo addressing can be applied to the effective address (EA) calculation associated with any W register. It is important to realize that the address boundaries checks look for addresses less than or greater than the upper (for incrementing buffers) and lower (for decrementing buffers) boundary addresses (not just equal to). Address changes may therefore jump over boundaries and still be adjusted correctly.

TABLE 176 -- MODCON, Modulo & Bit Reversed addressing Control Register
(0xxxxx)

Upper Half:

<u>R/W-0</u>	<u>R/W-0</u>	<u>U</u>	<u>U</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>
<u>XModeN</u>	<u>YModeN</u>	<u>:</u>	<u>:</u>	<u>BWM3</u>	<u>BWM2</u>	<u>BWM1</u>	<u>BWM0</u>
<u>bit 15</u>							<u>bit 8</u>

Lower Half:

<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>
<u>YWM3</u>	<u>YWM2</u>	<u>YWM1</u>	<u>YWM0</u>	<u>XWM3</u>	<u>XWM2</u>	<u>XWM1</u>	<u>XWM0</u>
<u>bit 7</u>							<u>bit 0</u>

15 XModeN: X AGU Modulus Addressing Enable
1 = X AGU Modulus Addressing enabled
0 = X AGU Modulus Addressing disabled

14 YModeN: Y AGU Modulus Addressing Enable
1 = Y AGU Modulus Addressing enabled
0 = Y AGU Modulus Addressing disabled

13 Unused

12 Unused

BWM: X AGU Register Select for Bit Reversed Addressing
0000 = W0 selected for bit reversed addressing

11-8 11
1110 = W14 selected for bit reversed addressing
1111 = W15 bit reversed addressing disabled

YWM: Y AGU W Register Select for Modulo Addressing
0000 = W0 selected for modulo addressing

7-4 11
1110 = W14 selected for modulo addressing
1111 = W15 modulo addressing disabled

XWM: X AGU W Register Select for Modulo Addressing
0000 = W0 selected for modulo addressing

3-0 11
1110 = W14 selected for modulo addressing
1111 = W15 modulo addressing disabled

Legend

R = Readable bit W = Writable bit
-n = Value at POR 1 = bit is set

U = Unimplemented bit, read as '0'
0 = bit is cleared

x = bit is unknown

TABLE 177 -- XMODSRT, X AGU Modulo addressing Start Register (XXXX h)Upper Half:

<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>
<u>XS15</u>	<u>XS14</u>	<u>XS13</u>	<u>XS12</u>	<u>XS11</u>	<u>XS10</u>	<u>XS9</u>	<u>XS8</u>
<u>bit 15</u>							<u>bit 8</u>

Lower Half:

<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>
<u>XS7</u>	<u>XS6</u>	<u>XS5</u>	<u>XS4</u>	<u>XS3</u>	<u>XS2</u>	<u>XS1</u>	<u>XS0</u>
<u>bit 7</u>							<u>bit 0</u>

15-0 XS: X AGU Modulo Addressing Start Address

Legend

R = Readable bit
-n = Value at POR

W = Writable bit
1 = bit is set

U = Unimplemented bit, read as '0'
0 = bit is cleared

x = bit is unknown

Modulo Addressing OperationTABLE 178 -- XModeND, X AGU Modulo addressing END Register (XXXX h)Upper Half:

<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>
<u>XE15</u>	<u>XE14</u>	<u>XE13</u>	<u>XE12</u>	<u>XE11</u>	<u>XE10</u>	<u>XE9</u>	<u>XE8</u>
<u>bit 15</u>							<u>bit 8</u>

Lower Half:

<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>
<u>XE7</u>	<u>XE6</u>	<u>XE5</u>	<u>XE4</u>	<u>XE3</u>	<u>XE2</u>	<u>XE1</u>	<u>XE0</u>
<u>bit 7</u>							<u>bit 0</u>

15-0 XE: X AGU Modulo Addressing End Address

Legend

R = Readable bit
-n = Value at POR

W = Writable bit
1 = bit is set

U = Unimplemented bit, read as '0'
0 = bit is cleared x = bit is unknown

TABLE 179 -- YMODSRT, Y AGU Modulo addressing Start Register (XXXX h)Upper Half:

<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>
<u>YS15</u>	<u>YS14</u>	<u>YS13</u>	<u>YS12</u>	<u>YS11</u>	<u>YS10</u>	<u>YS9</u>	<u>YS8</u>
<u>bit 15</u>							<u>bit 8</u>

Lower Half:

<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>
<u>YS7</u>	<u>YS6</u>	<u>YS5</u>	<u>YS4</u>	<u>YS3</u>	<u>YS2</u>	<u>YS1</u>	<u>YS0</u>
<u>bit 7</u>							<u>bit 0</u>

15-0 YS: Y AGU Modulo Addressing Start Address

Legend

R = Readable bit
-n = Value at POR

W = Writable bit
1 = bit is set

U = Unimplemented bit, read as '0'
0 = bit is cleared x = bit is unknown

TABLE 180 -- YModeND, Y AGU Modulo addressing END Register (XXXX h)Upper Half:

<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>
<u>YE15</u>	<u>YE14</u>	<u>YE13</u>	<u>YE12</u>	<u>YE11</u>	<u>YE10</u>	<u>YE9</u>	<u>YE8</u>
<u>bit 15</u>							<u>bit 8</u>

Lower Half:

<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>
<u>YE7</u>	<u>YE6</u>	<u>YE5</u>	<u>YE4</u>	<u>YE3</u>	<u>YE2</u>	<u>YE1</u>	<u>YE0</u>
<u>bit 7</u>							<u>bit 0</u>

15-0YE: X AGU Modulo Addressing End AddressLegendR = Readable bitW = Writable bitU = Unimplemented bit, read as '0'-n = Value at POR1 = bit is set0 = bit is clearedx = bit is unknownModulo Addressing Restrictions

[0368] As stated above, for an incrementing buffer the circular buffer start address (lower boundary) is arbitrary but must be at a 'zero', power of two boundary. For a decrementing buffer, the circular buffer end address is arbitrary but must be at a 'ones' boundary. With this scheme, there are no restriction regarding how much an EA calculation can exceeds the address boundary being checked, and still be successfully corrected. Once configured, the direction of successive addresses into a buffer cannot be changed. Although all EA's will continue to be generated correctly irrespective of offset sign, only one address boundary is checked for each type of buffer. Accessing an incrementing buffer with a decrementing address could result in the address decrementing through the start address. If this occurs, an out of range address will be detected but the address wrap operation will fail unless the end address is on a 'ones' address boundary (because the addition is simplified to an OR operation). For example, if the start address = 0x2000, end addresses that will support a bi-directional buffer include 0x200F, 0x203F or any modulo 2 length buffer. As similar augment applies to accessing a decrementing buffer with an incrementing address.

Modulo Addressing Timing

[0369] Modulo addressing can operate on both source and destination operands (i.e. for data reads and writes). Consequently, it must meet timing for the standard instruction cycle timing. Ideally, all AGU adder results should be stable by the end of Q1 (for reads and stack writes) or Q3 (for writes or stack reads). Effective address selection should occur on rising Q2 or Q4. The W address register update (when required) should occur during Q2.

[0370] Alternatively, each AGU could be built as an asynchronous block allowing the address calculation and selection to ripple through. However, it is highly likely that this will result in many spurious address transitions which could effect power consumption if allowed to propagate too far.

Bit Reversed Addressing

[0371] Bit reversed addressing is intended to simplify data re-ordering for radix-2 FFT algorithms. It is supported by the X AGU only. The carry propagation direction for a bit reversed EA calculation is changed to most significant bit to least significant bit. The modifier (a constant value or register contents) must also be regarded as having its bit order reversed. For example, for a 16 entry buffer (words & byte data size implications are discussed later), the address pointer and result are bit re-ordered as shown in Figure 67.

[0372] This example shows a pointer being incremented by one by an adder with a conventional carry direction. The modifier is presented in normal bit order (ls-bit to the right). The address pointer is a bit reversed EA and is presented in reversed bit order (LS-bit to the left). The address and result must be flipped around a pivot point in the middle of the address length in order for this to work with a conventional adder. The problem

arises when the buffer length is a variable which makes the bit swap operation unreasonably complex (the pivot point varies). An alternative is to keep the address source and destination in reversed order and use a bit reversed modifier with a reversed carry adder as shown in Figure 68. The net result is the same but the only operand requiring reversal is the modifier. As this is a constant, the reversed value does not need to be created for each calculation.

[0373] Table 181 shows the result of traversing the entire buffer, starting at address 0. Other modifier values will produce a bit-reversed address sequence, but only this one is reported to be of any real use.

TABLE 181 -- Bit Reversed Address Sequence (16-Entry)

<u>Bit Reversed Address</u>			
<u>A0</u>	<u>A1</u>	<u>A2</u>	<u>A3</u>
<u>0</u>	<u>0</u>	<u>0</u>	<u>0</u>
<u>1</u>	<u>0</u>	<u>0</u>	<u>0</u>
<u>0</u>	<u>1</u>	<u>0</u>	<u>0</u>
<u>1</u>	<u>1</u>	<u>0</u>	<u>0</u>
<u>0</u>	<u>0</u>	<u>1</u>	<u>0</u>
<u>1</u>	<u>0</u>	<u>1</u>	<u>0</u>
<u>0</u>	<u>1</u>	<u>1</u>	<u>0</u>
<u>1</u>	<u>1</u>	<u>1</u>	<u>0</u>
<u>0</u>	<u>0</u>	<u>0</u>	<u>1</u>
<u>1</u>	<u>0</u>	<u>0</u>	<u>1</u>
<u>0</u>	<u>1</u>	<u>0</u>	<u>1</u>
<u>1</u>	<u>1</u>	<u>0</u>	<u>1</u>
<u>0</u>	<u>0</u>	<u>1</u>	<u>1</u>
<u>1</u>	<u>0</u>	<u>1</u>	<u>1</u>
<u>0</u>	<u>1</u>	<u>1</u>	<u>1</u>
<u>1</u>	<u>1</u>	<u>1</u>	<u>1</u>

[0374] Bit reversed addressing is only supported by the X AGU. The address adder carry reverse signal (see Figure 63) is asserted when:

1. XWB (W register selection) in the XMOD register is any value other than 15 (it is assumed that nobody will ever want to bit reverse address the stack) and
2. the BREN bit is set in the XBREV register and
3. the addressing mode is register in direct with post-increment.

[0375] XB<14:0> is the bit reversed address modifier which is typically a constant, indirectly representing the size of the FFT data buffer. The XB values required to provide the correct bit reversal 'pivot' points for various size buffers are shown in Table 182. All bit reversed EA calculations assume word size data (LS-bit of every EA is always clear). The XB value is scaled accordingly to generate compatible (byte) addresses.

TABLE 182 -- Address Modifier Values

<u>Buffer Size</u> <u>(words)</u>	<u>12-bit Bit Reversed Address Modifier</u> <u>(XB)</u>	<u>XB Scaled for Word Sized Data</u>
<u>32768</u>	<u>0x4000</u>	<u>0x8000</u>
<u>16384</u>	<u>0x2000</u>	<u>0x4000</u>
<u>8192</u>	<u>0x1000</u>	<u>0x2000</u>
<u>4096</u>	<u>0x0800</u>	<u>0x1000</u>
<u>2048</u>	<u>0x0400</u>	<u>0x0800</u>
<u>1024</u>	<u>0x0200</u>	<u>0x0400</u>
<u>512</u>	<u>0x0100</u>	<u>0x0200</u>
<u>256</u>	<u>0x0080</u>	<u>0x0100</u>
<u>128</u>	<u>0x0040</u>	<u>0x0080</u>
<u>64</u>	<u>0x0020</u>	<u>0x0040</u>
<u>32</u>	<u>0x0010</u>	<u>0x0020</u>
<u>16</u>	<u>0x0008</u>	<u>0x0010</u>
<u>8</u>	<u>0x0004</u>	<u>0x0008</u>
<u>4</u>	<u>0x0002</u>	<u>0x0004</u>
<u>2</u>	<u>0x0001</u>	<u>0x0002</u>

[0376] As can be seen from Figure 68, requiring that both the address modifier (constant) and the address in the W pointer are always in bit reversed format simplifies the hardware. Adding two bit reversed values though the adder with carry reversed enabled, will produce the correct bit reversed result. When enabled, bit reversed addressing will only be executed with register indirect with post increment addressing and word sized data. It will not function for all other addressing modes or byte sized data (normal addresses will be generated). When bit reversed addressing is active, the W address pointer will always be added to the address modifier (XB) and the offset associated with the register indirect addressing mode will be ignored. In addition, as word sized data is a requirement, the LS-

bit of the EA is ignored (and always clear). An example word swap using bit reversed addressing is:

```
MOV [W9], W0
MOV [W8], [W9]+
MOV W0, [W8]+
```

See in particular Figures 69 and 70.

TABLE 183 -- XBREV, X AGU Bit Reversal addressing Control Register (xxxx h)

Upper Half:

<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>
<u>BREN</u>	<u>XB14</u>	<u>XB13</u>	<u>XB12</u>	<u>XB11</u>	<u>XB10</u>	<u>XB9</u>	<u>XB8</u>
<u>bit 15</u>							<u>bit 8</u>

Lower Half:

<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>
<u>XB7</u>	<u>XB6</u>	<u>XB5</u>	<u>XB4</u>	<u>XB3</u>	<u>XB2</u>	<u>XB1</u>	<u>XB0</u>
<u>bit 7</u>							<u>bit 0</u>

BREN: Bit Reversed Addressing (X AGU) Enable

15 1 = Bit Reversed Addressing enabled

0 = Bit Reversed Addressing disabled

14-0 XB<14:0>: X AGU bit reversed Modifier

e.g. XB<14:0> = 0x0080; modifier for a 128 point radix-2 FFT

Legend

R = Readable bit
-n = Value at POR

W = Writable bit
1 = bit is set

U = Unimplemented bit, read as '0'

0 = bit is cleared

x = bit is unknown

[0377] Many applications require significant amounts of fixed data (e.g. MELP) which can only be held in non-volatile memory. This data can also exceed the 32K word limit of data space memory. Consequently, this data will have to reside in on-chip program FLASH, ROM or in external program space. In order to accommodate this requirement, two addressing options are provided.

1. The table instructions allows direct movement of word and byte data respectively between program and data space without passing through an intermediate register.
2. The upper part of data space may be configured to map into a 16K word segment of program space.

[0378] The operation of these addressing options is discussed elsewhere in this specification. The following sections revisit the table instructions, in particular the addressing modes supported.

Table Instruction Operation

[0379] There are four 'table' instructions as shown in Table 184 that operate with Mode 2 addressing modes for both operand source and destination. They operate in a manner similar to that for data space access except that the EA for program space (source or destination) is concatenated with a 8-bit page register, TABPAG<7:0> to create a 24-bit address. All table instructions treat the program memory as 16-bit wide, byte addressable (i.e. same as data space). Program space EA[24:1] forms the 24-bit program memory address and the EA[0] becomes a byte select bit. The TBLRDL and TBLWTL instructions are dedicated to accessing the LS program word.

[0380] The program word is viewed as a 32-bit entity which consists of a 24-bit program word plus an 8-bit 'phantom' byte (MS-byte). This allows TBLRDH and TBLWTH instructions (which are dedicated to accessing the MS program word) to maintain orthogonality with TBLRDL and TBLWTL. For TBLRDH and TBLWTH instructions, EA[0] remains a byte select bit but physical memory is only present in the LS-byte (EA[0]=0). A byte read of the MS-byte (EA[0]=1) will return 0x00.

Table Read Operation

[0381] The program memory is always read as 24-bit long words. The LS-bit of the EA is used by the TBLRDL and TBLWTL (if required) to select required byte of the LS program word. Table 184 indicates which instruction and data width will access the various parts of the program word.

TABLE 184 -- Table Instruction Summary

<u>Instruction</u>	<u>EA[0]</u>	<u>Program Space</u>	
		<u>Source</u>	<u>Destination</u>
<u>TBLRDH.w 1</u>	<u>x</u>	<u>[EAsrc]<31:16></u>	<u>[EAdst]<15:0></u>
<u>TBLRDH.b 0</u>	<u>0</u>	<u>[EAsrc]<16:23></u>	<u>[EAdst]<7:0></u>
<u>TBLRDH.b 1</u>	<u>1</u>	<u>[EAsrc]<31:24></u>	<u>[EAdst]<7:0></u>
<u>TBLRDL.w x</u>	<u>x</u>	<u>[EAsrc]<15:0></u>	<u>[EAdst]<15:0></u>
<u>TBLRDL.b 0</u>	<u>0</u>	<u>[EAsrc]<7:0></u>	<u>[EAdst]<7:0></u>
<u>TBLRDL.b 1</u>	<u>1</u>	<u>[EAsrc]<15:8></u>	<u>[EAdst]<7:0></u>

MS-byte read will return 0x00

[0382] TBLRDH.w reads a data word from [EAsrc]<31:16>, though [EAsrc]<31:24> will equal 0x00. TBLRDH.b reads a data byte from [EAsrc]<31:24> (always equal to 0x00) or [EAsrc]<16:23> based on the state of EA[0]. The data byte is transferred into destination EA[7:0].

[0383] TBLRDL.w reads a data word from [EAsrc]<15:0>. TBLRDL.b reads a data byte from [EAsrc]<15:0> or [EAsrc]<7:0> based on the state of EA[0]. The data byte is transferred into destination EA[7:0].

[0384] For most applications, it is assumed that only the LS word of the program word will be used for data storage. The MS byte of the program word would then typically contain an illegal instruction trap to prevent the machine from ever inadvertently attempting to execute data. However, TBLRDH is provided to allow the use of all program memory for data storage if desired.

Table Writes

Mode 2 Addressing for Program Space

[0385] Mode 2 determines the addressing mode for the operand source/destination in program space or the operand source/destination from data space, depending upon instruction requirements. It follows the same definition for each encoding as Mode 1

except that it applies to only one operand. The Mode 1 signed 5-bit constant value mode makes little sense where Mode 2 is used, and is therefore not supported.

[0386] In summary, Mode 2 for program space data accesses supports the addressing mode shown in Table 185. Mode 2 Submode 0 is meaningless for TBLRD source and TBLWT destination operands as the program memory must be addressed with a pointer. The following addressing mode descriptions are for table read operations.

TABLE 185 -- Mode 2 Addressing Mode Definition (Program Space)

<u>Mode 2</u>	<u>Function</u>	<u>Function</u>	<u>Description</u>
<u>Bit Encoding</u>	<u>(Source)</u>	<u>(Destination)</u>	
<u>000</u>	<u>EA = Wsrc 1</u>	<u>EA = Wdst 2</u>	<u>Register direct</u>
<u>001</u>	<u>EA = [Wsrc]</u>	<u>EA = [Wdst]</u>	<u>Register indirect</u>
<u>010</u>	<u>EA = [Wsrc]-= 1</u>	<u>EA = [Wdst]-= 1</u>	<u>Register indirect post-decremented</u>
<u>011</u>	<u>EA = [Wsrc]+= 1</u>	<u>EA = [Wdst]+= 1</u>	<u>Register indirect post-incremented</u>
<u>100</u>	<u>EA = [Wsrc-=1]</u>	<u>EA = [Wdst-=1]</u>	<u>Register indirect pre-decremented</u>
<u>101</u>	<u>EA = [Wsrc+=1]</u>	<u>EA = [Wdst+=1]</u>	<u>Register indirect pre-incremented</u>
<u>110</u>	<u>Unused</u>	<u>Unused</u>	
<u>111</u>	<u>Unused</u>	<u>Unused</u>	

Note, this is not meaningful for TBLRD or TBLWT instructions

Mode 2, Register Direct

[0387] Addressing Mode 2, Submode 0 is register direct. The implied effective address is the memory mapped address of register Wdst. The table read result is written to Wdst as shown in Figure 71. Wdst is accessed through addressing its memory mapped image. Note that register direct for the operand source of a table read, and the operand destination for a table write has no meaning. The X AGU would generate an EA which would address the memory mapped version of Wsrc or Wdst. When concatenated with the TABPAG register, this address will be the same but within a program space page.

Mode 2, Register Indirect

[0388] Addressing Mode 2, Submode 1 is register indirect. The effective address contained in register Wsrc points to the operand as shown in Figure 72, or Wdst points to the result destination as shown in Figure 73. For table read instructions, TABPAG<7:0> is concatenated onto the source EA to form the 24-bit program space EA.

Mode 2, Register Indirect with Post Decrement

[0389] Addressing Mode 2, Submode 2 is register indirect with post decrement. The effective address contained in register Wsrc points to the operand, or the effective address contained in register Wdst points to the result destination. Wsrc or Wdst is then post decremented as shown in Figure 74 and Figure 75.

Mode 2, Register Indirect with Post Increment

[0390] Addressing Mode 2, Submode 3 is register indirect with post increment. The effective address contained in register Wsrc points to the source operand, or the effective address contained in register Wdst points to the result destination. Wsrc or Wdst are then incremented as shown in Figure 76 and Figure 77.

Mode 2, Register Indirect with Pre Decrement

[0391] Addressing Mode 2, Submode 4 is register indirect with pre decrement. Register Wsrc or Wdst is decremented to form the effective address which points to the operand as shown in Figure 78 and Figure 79.

Mode 2, Register Indirect with Pre Increment

[0392] Addressing Mode 2, Submode 5 is register indirect with pre increment. Register Wsrc or Wdst is incremented to form the effective address which points to the operand as shown in Figure 80 and Figure 81.

[0393] Figure 82 shows a timing diagram illustrating a XOR, the SUBR, the SUBR, the SUB B, the SUB, the MOVE, the IOR, the AND, the ADDC and the ADD operations. Figure 83 shows a timing diagram illustrating a XORLS, the SUBRLS, the SUBLS, the SUBBRLS, the SUBLS, the IORLS, the ANDLS, the ADCCLS and the ADDCLS operations. Figure 84 shows a timing diagram illustrating a COR, the INC2, the DEC2, the DEC COM, the NEG and the NCTM operations. Figure 85 shows a timing diagram illustrating a ASR, the LSR, the ZE, the SE, the SL, the RLC, the RLNC, the RRC and the RRNC operation.

[0394] Figure 86 shows a timing diagram illustrating a CPB and the CP operations. Figure 87 shows a timing diagram illustrating a CP1 and the CP0 operations. Figure 88 shows a timing diagram illustrating a CPBLS and the CPBLS operations. Figure 89 shows a timing diagram illustrating a XORLW, the SUBLW, the SUBBBLW, the MOVLW, the MOVL, the IORLW, the ANDLW, the ADDLW and the ADDCLW operations. Figure 90 shows a timing diagram illustrating a ASRF, the SLF, the LSRF, the RRNCF, the RRCE, the RLNCF, the RLCE, the XORWE, the SUBWS, the SUBBWS, the SUBFW, the SUBDFW, the MOVFW, the MOV, the IORWV, the ANDWE, the ADDWFC and the ADDWF operations.

[0395] Figure 91 shows a timing diagram illustrating a CPEB, the CPE1, the CPF0 and the CPF operations. Figure 92 shows a timing diagram illustrating a INCE, the DECE, the NEGE, the SETE, the COMF and the CLRf operations. Figure 93 shows a timing diagram illustrating a CPFSEQ, the FPESGT, the CPESLT and the CPESNE operations. Figure 94 shows a timing diagram illustrating an INCESNZ, the INCESA, the DECFSNZ, and the DECFSZ operations.

[0396] Figure 95 shows a timing diagram illustrating a SWAP operation. Figure 96 shows a timing diagram illustrating a STW operation. Figure 97 shows a timing diagram illustrating a EXCH operation. Figure 98 shows a timing diagram illustrating a BSW operation. Figure 99 shows a timing diagram illustrating a BTSTW operation.

[0397] Figure 100 shows a timing diagram illustrating a BCLRE, the BTSTSE, the BTSTF, the BTGF and BSETF operations. Figure 101 shows a timing diagram illustrating a BSET, the BTG, the BTST, the BTSTS, and the BCLR operations. Figure 102 shows a timing diagram illustrating a BTSS, the BTSC, the BTFSC and the BTFSS operations. Figure 103 shows a timing diagram illustrating a TBLRDH and the TBLRDL operations. Figure 104 shows a timing diagram illustrating a TBLWTH and the TBLWTL operations.

[0398] Figure 105 shows a timing diagram illustrating a LDQW operation. Figure 106 shows a timing diagram illustrating a LDDW operation. Figure 107 shows a timing diagram illustrating a STQW operation. Figure 108 shows a timing diagram illustrating a STDW operation. Figure 109 shows a timing diagram illustrating a MULS, the MULSU, the MULSULS, the MULU, the MULULS and the MULUS operations. Figure 110 shows a timing diagram illustrating a MULWF operation. Figure 111 shows a timing diagram illustrating an ALL BRANCHES operation.

[0399] Figure 112 shows a timing diagram illustrating a BRAW operation. Figure 113 shows a timing diagram illustrating a RCALL, and the RCALLW operations. Figure 114 shows a timing diagram illustrating a CALLW operation. Figure 115 shows a timing diagram illustrating a CALL operation. Figure 116 shows a timing diagram illustrating a GOTOW operation. Figure 117 shows a timing diagram illustrating a GOTO operation.

[0400] Figure 118 shows a timing diagram illustrating a LNK operation. Figure 119 shows a timing diagram illustrating a ULNK operation. Figure 120 shows a timing diagram illustrating a DAW operation. Figure 121 shows a timing diagram illustrating a SCRATCH operation. Figure 122 shows a timing diagram illustrating a ITCH operation. Figure 123 shows a timing diagram illustrating a PUSH operation. Figure 124 shows a timing diagram illustrating a POP operation. Figure 125 shows a timing diagram illustrating a LDW operation. Figure 126 shows a timing diagram illustrating a TRAP operation. Figure 127 shows a timing diagram illustrating a DISI operation. Figure 128 shows a timing diagram illustrating a LDW operation. Figure 129 shows a timing diagram illustrating a DO and the DOW operations.

[0401] Figure 130 shows a timing diagram illustrating a DO and the DOW operations continued. Figure 131 shows a timing diagram illustrating a MAC, the CLRAC, the EDAC, the SQRAC and the MOVSAC operations. Figure 132 shows a timing diagram illustrating a SQR, the ED, the MPY, the MPYN and the MSC operations. Figure 133 shows a timing diagram illustrating a LSRW, the LSRK, the ASRK, the ASRW, the SLW and the SLK operations. Figure 134 shows a timing diagram illustrating a ADDAB, the NEGAB and the SUBAB operations. Figure 135 shows a timing diagram illustrating an ADDAC operation.

[0402] Figure 136 shows a timing diagram illustrating a LAC operation. Figure 137 shows a timing diagram illustrating a SAC and the SAC.R operations. Figure 138 shows a timing diagram illustrating a SFTACK and the SFTAC operations. Figure 139 shows a timing diagram illustrating a RETURN, the RE and the TFIE operations. Figure 140 shows a timing diagram illustrating a MSLK, the MSRK, the MSLW and the MSRW

operations. Figure 141 shows a timing diagram illustrating a FBCL, the FBCR, the FFOL, the FFOR, the FFIL and the FFIR operations. Figure 142 shows a timing diagram illustrating a RETLW operation. Figure 143 shows a timing diagram illustrating a REPEAT and the REPEAT W operations. Figure 144 shows a timing diagram illustrating a REPEAT and the REPEAT W operations continued.

Architectural Description

[0403] The foregoing illustrative example utilizes the disclosure provided above for various descriptions. The illustrative example of the central processing unit disclosed herein is a 16-bit (data) modified Harvard architecture with a greatly enhanced instruction set including significant support for digital signal processing (DSP). The foregoing description is better understood with reference to Figures 145-172. Specifically, Figure 145 shows a block diagram illustrating a CPU Core. Figure 146 shows a block diagram illustrating data alignment. Figure 147 shows a block diagram illustrating a Data Space Memory Map Example. Figure 148 shows a block diagram illustrating a data space for a microcontroller and digital signal processor instructions.

[0404] Figure 149 shows a block diagram illustrating a data space window into the program space operation. Figure 150 shows a block diagram illustrating a PS Data Read-Through DS operation. Figure 151 shows a block diagram illustrating a PS Data Read-Through DS within a REPEAT loop operation. Figure 152 shows a block diagram illustrating a data access operation from program space address generation.

[0405] Figure 153 shows a block diagram illustrating an instruction fetch. Figure 154 shows a block diagram illustrating a program space memory map. Figure 155 shows a block diagram illustrating a program data table access. Figure 156 shows a block diagram

illustrating program data table access. Figure 157 shows a block diagram illustrating HEX file compatibility. Figure 158 is a basic core timing diagram. Figure 159 shows a timing diagram illustrating a clock/instruction cycle. Figure 160 is a flow chart illustrating a REPEAT[W] loop functional flow. Figure 161 shows a block diagram illustrating a REPEAT[W] instruction pipeline Flow. Figure 162 shows a block diagram of a Do Loop hardware operation. Figure 163 is timing diagram of a DO loop entry operation. Figure 164 shows a timing diagram illustrating a DO loop continuation operation. Figure 165 shows a timing diagram illustrating a DO Continue with Branch to Last Instruction operation. Figure 166 shows a timing diagram illustrating a DO loop EXIT operation. Figure 167 is a flow chart illustrating a DO and REPEAT operation.

[0406] Figure 168 shows a block diagram illustrating an Uninitialized W Register Trap operation. Figure 169 shows a block diagram illustrating a Stack Pointer Overflow & Underflow Trap operation. Figure 170 shows a timing diagram illustrating a Stack Timing of a PC PUSH CALL operation. Figure 171 shows a timing diagram illustrating a Stack Timing of a PC POP RETURN operation. Figure 172 shows a block diagram illustrating a CALL stack frame.

Core Overview

[0407] The core has a 24-bit instruction word, with a variable length opcode field. The PC is 24-bits wide (with the LS-bit always clear) addressing up to 8M long words (23-bits). An 'C18-like' instruction prefetch mechanism is used to help maintain throughput. Deeper levels of pipelining have been intentionally avoided to maintain good real-time performance. Unconditional overhead free program loop constructs are supported using the DO and REPEAT instructions, both of which are interruptable at any point.

[0408] The working register array has been extended to 16 x 16-bit registers, each of which can act as data, address or offset registers. One working register (W15) operates as a software stack for interrupts and calls.

[0409] The data space is 32K words of word or byte addressable space which is split into two blocks referred to as X and Y data memory. Each block has its own independent Address Generation Unit (AGU). Most instructions operate solely through the X memory AGU which will make it appear as one linear space encompassing all data space. The MAC class of DSP instructions will operate through both the X and Y AGUs, splitting the data address space into two parts. The X and Y data space boundary is arbitrary and defined through the address decode of each memory array. See Figure 63, Figure 64 and accompanying description.

[0410] The upper 32K bytes of data space memory can optionally be mapped into the lower half (user space) of program space at any 16K program word boundary defined by the 8-bit Data Space Program PAGE (DSPPAG) register. This lets any instruction to access program space as if it were data space (other than the additional access cycle it consumes) plus allows external RAM hooked onto the external program space to be mapped into data space, effectively providing an external data space bus.

[0411] Overhead free circular buffers (modulo addressing) are supported in both X and Y address spaces. They are intended to remove the loop overhead for DSP algorithms but X modulo addressing can be universally applied using any instructions.

[0412] The X AGU also supports bit reverse addressing to greatly simplify input or output data reordering for radix-2 FFT algorithms.

[0413] The core supports inherent (no operand), relative, literal, memory direct and four groups of addressing modes (MODE 1, MODE 2, MODE 3 and MODE 4) for register direct and register indirect modes. Each group offers up to six addressing modes. Instructions are associated with predefined addressing modes depending upon their functional requirements.

[0414] For most instructions, the core is capable of executing a data (or program data) memory read, a working register (data) read, a data memory write and a program (instruction) memory read per instruction cycle. As a result, three operand instructions can be supported, allowing $A+B=C$ operations to be executed in a single cycle.

[0415] A DSP engine has been included to significantly enhance the core arithmetic capability and throughput. It features a high speed 16-bit by 16-bit multiplier, a 40-bit ALU, two 40-bit saturating accumulators and a 40-bit bidirectional barrel shifter. The barrel shifter is capable of shifting a 40-bit value up to 15 bits right or up to 16-bits left in a single cycle. The DSP instructions operate seamlessly with all other instructions and have been designed for optimal real-time performance. The MAC class of instructions can concurrently fetch two data operands from memory while multiplying two W registers. This requires that the data space be split for these instructions and linear for all others. This is achieved in a transparent and flexible manner through dedicating certain working registers to each address space for the MAC class of instructions.

[0416] The core features a vectored exception scheme with 15 individually prioritized vectors. The exceptions consist of reset, seven traps and eight interrupts. One interrupt level may be selected (typically the highest one) to execute as a fast (1 cycle entry, 1 cycle exit) interrupt. This function is actually an extension of the logic required to allow a

REPEAT instruction loop to be interrupted which can significantly reduce latency in some application. A block diagram of the core is shown in Figure 145.

Compiler Driven Enhancements

[0417] In addition to DSP performance requirements, the core architecture was strongly influenced by recommendations which would lead to a more efficient (code size and speed) C compiler.

[0418] For most instructions, the core is capable of executing a data (or program data) memory read, a working register (data) read, a data memory write and a program (instruction) memory read per instruction cycle. As a result, three operand instructions can be supported, allowing $A+B=C$ operations to be executed in a single cycle. Instruction addressing modes are significantly more flexible than those of other processors, and are matched closely to compiler needs. The working register array has been extended to 16 x 16-bit registers, each of which can act as data, address or offset registers. One working register (W15) operates as a software stack for interrupts and calls.

[0419] Linear indirect access of all data space is possible, plus the memory direct address range has been extended to 8Kbytes (256bytes in C18). This together with the addition of 16-bit direct address LOAD and STORE instructions has allowed the C18 data space memory banking scheme to be eliminated. Linear indirect access of 32K word (64K byte) pages within program space (user and test space) is possible using any working register via new table read and write instructions. Part of data space can be mapped into program space, allowing constant data to be accessed as if it were in data space.

Instruction Fetch Mechanism

[0420] The core does not support an instruction pipeline. A pre-fetching mechanism accesses instruction a cycle ahead to maximize available execution time. Most instructions execute in a single cycle. Exceptions are:

1. Flow control instructions and interrupts where the ISR (instruction register) and pre-fetch buffer must be flushed and refilled.
2. Instructions where one operand is to be fetched from program space (using any method). These operations consume 2 cycles (with the notable exception of the MAC class of DSP instructions executed within a REPEAT loop which executes in 1 cycle).

[0421] Most instructions access data as required during instruction execution. Instructions which utilize the multiplier array must have data available at the beginning of the instruction cycle. Consequently, this data must be prefetched, usually by the preceding instruction, resulting in a simple out of order data processing model.

Data Address Space

[0422] The core features one program space and two data spaces. The data spaces can be considered either separately (for some DSP instructions) or together as one linear address range (for MCU instructions). The data spaces are accessed using two Address Generation Units (AGUs) and separate data paths.

Data Spaces

[0423] The X AGU is used by all instructions and supports all addressing modes. It also supports modulo and bit reversed addressing for any instructions subject to addressing mode restrictions (see [See Modulo and Bit Reversed Addressing Controller]). The X data path is the return data path for all single data space access instructions.

[0424] The Y AGU and data path are used in concert with the X AGU by the MAC class of instructions to provide two concurrent data read paths. No writes occur across the Y-bus. This class of instructions dedicate two W register pointers, W6 and W7, to always

operate through the Y AGU and address Y data space independently from X data space.

Note that during accumulator write-back, the data address space is considered combined X and Y, so the write will occur across the X-bus. Consequently, it can be to any address irrespective of where the EA is directed.

[0425] The Y AGU only supports MODE 4 post modification addressing modes associated with the MAC class of instructions. It also supports modulo addressing for automated circular buffers. Of course, all other instructions can access the Y data address space through the X AGU when it is regarded as part of the composite linear space.

[0426] The boundary between the X and Y data spaces is arbitrary and is defined by the memory address decode only (the CPU has no knowledge of the physical location of X or Y memory). The boundary is not user programmable but may change from variant to variant. Obviously, to present a linear data space to the MCU instructions, the address spaces of X and Y data spaces must be contiguous but this is not an architectural necessity. Note that any memory located between 0x8000 and 0xFFFF will not be accessible when program space visibility is enabled for this address space. It should be noted that as address space 0x8000 to 0xFFFF can map to a single memory in program space, it may need to be assigned to either X or Y space (but not both since concurrent accesses from the same space are not possible).

[0427] All (effective addresses) are 16-bits wide and point to bytes within the data space to facilitate backward compatibility with previous processors. Consequently, the data space address range is 64K bytes or 32K words.

Data Space Width

[0428] The core data width is 16-bits. All internal registers and data space memory are organized as 16-bits wide (some CPU registers are not 16-bits wide, see Figure 5). Data space memory is organized in byte addressable, 16-bit wide blocks. Byte addressability requires independent byte write signals for upper and lower bytes.

Data Alignment

[0429] To help maintain backward compatibility and improve data space memory usage efficiency, the ISA supports both word and byte operations. Referring to Figure 146, data is aligned in data memory and registers as words, but all data space EAs resolve to bytes. Data byte reads will read the complete word which contains the byte, using the LS-bit of any EA to determine which byte to select. The selected byte is place onto the LS-byte of the X data path (no byte accesses are possible from the Y data path as the MAC class of instruction can only fetch words). That is, data memory and registers are organized as two parallel byte wide entities with shared (word) address decode but separate write lines. Data byte writes will only write to the corresponding side of the array or register which matches the byte address. For word accesses, the LS-bit of the EA is ignored.

[0430] Byte reads will always read the entire word, so mechanisms to clear or set peripheral status bits when read (e.g. quick flag clearing mechanisms) are not allowed. As a consequence of this byte accessibility, all effective address calculations (including those generated by the DSP operations which are restricted to word size) must be scaled to step through word aligned memory. For example, the core must recognize that post modified register indirect addressing mode, $[Ws] += 1$, will result in a value of $Ws+1$ for byte operations and $Ws+2$ for word operations.

[0431] All word accesses must be aligned (to an even address). Mis-aligned word data fetches are not supported so care must therefore be taken when mixing byte and word operations or translating from C18 code. Should a mis-aligned read or write be attempted, an address fault trap will forced. Depending upon where the fault occurred in the instruction cycle, the Q1/Q2 access (typically a read) and/or the Q3/Q4 access (typically a write) for the instruction underway will be inhibited, and the PC will not be incremented. The trap will then be taken, allowing the system and/or user to examine the machine state prior to execution of the address fault.

[0432] All byte loads into any W register are loaded into the LS-byte. The MS-byte is not modified. It should be noted that byte operations use the 16-bit ALU and can produce results in excess of 8-bits. However, to maintain C18 backwards compatibility, the ALU result from all byte operations is written back as a byte (i.e. MS byte not modified), and the status register is updated based only upon the state of the LS-byte of the result.

[0433] A sign extend (SE) instruction is provided to allow users to translate 8-bit signed data to 16-bit signed values. Alternatively, for 16-bit unsigned data, users can clear the MS-byte of any W register though executing a CLR.b instruction on the appropriate address.

[0434] Although most instructions are capable of operating on word or byte data sizes, it should be noted that the DSP and some other new instructions operate on words only.

Data Space Memory Map

[0435] The data space memory is split into two blocks, X and Y data space. A key element of this architecture is that Y space is a subset of X space, and is fully contained

within X space. In order to provide an apparent linear addressing space, X and Y space would typically have contiguous addresses (though this is not an architectural necessity).

[0436] When executing any instruction other than a MAC class one, the X block consists of the entire 64Kbyte data address space (including all Y addresses). When executing a MAC class of instruction, the X block consists of the entire 64Kbyte data address space less the Y address block for data reads (only). In other words, the full address space is available to all instructions other than the MAC class. During Q1/Q2 data reads, the MAC class of instructions extracts the Y address space from data space and addresses it using EA's sourced from W6 and W7. The remaining data space is referred to as X space but could more accurately be described as "X-Y" space, and is concurrently addressed using W4 and W5 during the same Q1/Q2 data read portion of the cycle. Both "X-Y" and Y address spaces are concurrently accessed only by the MAC class of instruction.

[0437] Note that it is the register number (and instruction class) that determine which address space is accessed for data reads and not the EA. Consequently, the data space partitioning of Y address space is arbitrary. In all cases, should an EA point to unoccupied space, all zeros will be returned. For example, although Y address space is visible by all non-MAC class instructions using any addressing mode, an attempt by a MAC instruction to fetch data from that space using W4 or W5 (X space pointers) will return 0x0000.

[0438] An example data space memory map is shown in Figure 147. Note again that the partition between each address space is arbitrary and determined by the memory decode. Both X and Y address generation units (AGUs) can generate any effective address

(EA) within a 64Kbyte range, however, EAs outside the physical memory provided will return all zeros.

[0439] An 8Kbyte access space is reserved in X address memory space between 0x0000 and 0x1FFF which is directly addressable via a 13-bit absolute address field within all memory direct instructions. The remaining X address space and all of the Y address space is addressable indirectly. The whole of X data space is additionally addressable using LDW and STW instructions which support memory direct addressing with a 16-bit address field.

Program Space Visibility from Data Space

[0440] The upper 32Kbytes of data space may optionally be mapped into any 16Kword program space page. This provides transparent access of stored constant data from X data space without the need to use special instructions (i.e. TBLRD, TBLWT instructions). Granularity of program space window may change, subject to conclusions of code security analysis.

[0441] This feature also allows the user to map the upper half of data space into an unused area of program memory and thus to the external bus (all unused internal addresses will be mapped externally). Through the placement of an external RAM at this address, external data space support is also provided. Data read and writes must therefore be supported to this address space. Note that the external address map is now essentially no longer strictly Harvard as program and data memory are combined.

[0442] Program space access through the data space occurs if the MS-bit of the data space EA is set and program space visibility is enabled by setting the PSV bit in the Core Control register ("CORCON"). Most of the CORCON function relate to DSP operation.

Depending upon FLASH setup and access time, the instruction may need to be at least partially pre-decoded during Q4 of the prior instruction. Even so, this will remain a critical path, as the source EA cannot be evaluated until the data write completes in the prior instruction.

[0443] Data accesses to this area will add an addition cycle to the instruction being executed since two program memory fetches will be required. The data is fetched in the first cycle, which, other than for some instruction decode, is essentially a NOP. The next instruction is prefetched in the second cycle while the current instruction completes execution (i.e. normal operation) as shown in Figure 150.

[0444] Furthermore, instructions executing from internal program memory but accessing external data memory RAM will sustain additional delay due to wait state insertion. Read-modify-write operations will sustain twice the delay. The External Bus Interface (EBI) definition is not complete at this time, however, it is expected that the device will be required to insert an even number of Q clocks into the instruction cycle between Q2 and Q3, and between Q4 and Q1 (of the next cycle) for external data space accesses.

[0445] Although not an architectural necessity, a typical data space configuration would define Y data space to be outside this re-mappable area, making the visible program space map to X data space. Y data space will typically contain state (variable) data for DSP operations, and must therefore be RAM. X data space will typically contain coefficient (constant) data which could be NVM or initialized RAM.

[0446] Although each transparent data space address will map directly into a program address (see Figure 152), only the lower 16-bits of the 24-bit program word are

used to contain the data. The upper 8-bits should be programmed to force an illegal instruction or software trap to maintain machine robustness.

[0447] For external accesses, data space would only require a 16-bit data path, with the trap instruction being automatically concatenated onto any 16-bit data reads.

[0448] The data space address is mapped into program memory as shown in Figure 152. Note that, by incrementing the PC by 2 for each program memory word, the LS 14 bits (15 bits for the TBLRD, TBLWT instructions) of data and program space addresses directly translate. The remaining bits are provided by the Data Space Program PAGE register, DSPPAG<7:0> as shown in Figure 152.

Data Pre-Fetch from Program Space within a REPEAT loop

[0449] When prefetching data resident in program space via the data space window from within a REPEAT loop, all iterations of the repeated instruction will reload the instruction from the Instruction Latch without re-fetching it, thereby releasing the program bus for a data prefetch as shown in Figure 151. In this example, the initial 2 data words for the first iteration of the instruction to be repeated (MACA) are fetched by a CLRACA instruction. As one of the words resides in program space, an additional cycle is required. The initial fetch of the MACA instruction is performed by the REPEAT instruction.

[0450] It is important to note that only the MAC class of instructions, which operate with prefetched data, will operate in this manner. All other instructions (e.g. MOV) which require data to be read by the end of Q2 will require the additional cycle in order to complete the data read prior to execution of the instruction during the second cycle.

Program Address Space

[0451] The program address space is 8M long words. It is addressable by a 24-bit value from either the PC, table instruction EA or data space EA when program space is mapped into data space as defined by Table 186. Note that the program space address is incremented by two between successive program words in order to provide compatibility with data space addressing. Consequently, the LS-bit of the program space address is always 0, resulting in 23-bits (8M) of address. Program space data accesses use the LS-bit of the program space address as a byte select (same as data space). Memory mapped or stacked PC may need to include the zero LS-bit.

[0452] The address space is split into two 4M long word spaces, one for user space the other for test and vector memory space as shown in Figure 154. When in user mode, program space access is restricted to the lower 4M long word space, 0x000000 to 0x7FFFFFFE for all accesses other than TBLRD/TBLWT which use TABPAG[7] to determine user or test space access. Exception vectors also reside in test space. While in user mode, the PC is inhibited from 'rolling over' into test space (i.e. PC[23] is always clear).

TABLE 186 -- Program Space Address Construction

<u>Access Type</u>	<u>Access Space</u>	<u>Program Space Address</u>				
		[23]	[22:16]	[15]	[14:1]	[0]
<u>Instruction Access</u>	<u>User</u>	<u>0</u>	<u>PC[23:1]</u>			<u>0</u>
<u>Instruction Access</u>	<u>Test</u>	<u>1</u>	<u>PC[23:1]</u>			<u>0</u>
<u>TBLRD/TBLWT</u>	<u>User/Test</u>	<u>TABPAG[7:0]</u>	<u>Data EA [15:0]</u>			
<u>DS Window into PS</u>	<u>User</u>	<u>0</u>	<u>DSPPAG[7:0]</u>	<u>Data EA [14:0]</u>		
<u>DS Window into PS</u>	<u>Test</u>	<u>Not allowed</u>				

[0453] The program memory width is 24-bits (long word). To support data storage and FLASH programming, the array must support both word wide access from bits 0-15 and byte wide access from bits 16-23. An instruction fetch example is shown in Figure 153. Note that incrementing PC[23:1] by one is equivalent to adding 2 to PC[23:0]. This architecture (internally) fetches 24-bit wide program memory. Consequently, instructions are always aligned. However, as the architecture is modified Harvard, data can also be present in program space.

[0454] There are two methods by which program space can be accessed - via special TABLE instructions or through the remapping of a 16Kword program space page into the upper half of data space (see Figure 154). The TBLRDL and TBLWTL instructions offers a direct method of reading or writing the LS word of any address within program space without going through data space which is preferable for some applications. The TBLRDH and TBLWTH instructions are the only method whereby the upper 8-bits of a program word can be accessed as data. Figure 152 shows how the EA is created for table operations.

Table instructions

[0455] A set of TABLE instructions are provided to move byte or word sized data to and from program space. The instructions are orthogonal even though the MS byte will always read zeros. See dsPIC Instruction Set DOS for more details.

1. TBLRDL: Table read low

Word: Read the LS word of the program address

P[15:0] maps to D[15:0]

Byte: Read one of the LS bytes of the program address

P[7:0] maps to D[7:0] when byte select=0;

P[15:8] maps to D[7:0] when byte select=1

2. TBLWRL: Table write low

See Program Memory DOS-00204

3. TBLRDH: Table read high

Word: Read the MS word of the program address

P[23:16] maps to D[7:0]; D[15:8] always = 0

Byte: Read one of the MS bytes of the program address

P[23:16] maps to D[7:0] when byte select=0;

D[7:0] will always = 0 when byte select=1

4. TBLWRH: Table write high

See Program Memory DOS-00204

Where:

P = program space long word; and

D = data space word.

[0456] The PC is incremented by two for each successive 24-bit program word. This allows program memory addresses to directly map to data space addresses as shown in Figure 155. Program memory can thus be regarded as two 16-bit word wide address spaces residing side by side, each with the same address range. TBLRDL and TBLWTL access the space which generates the LS data word, and TBLRDH and TBLWTH access the space which generates the MS data byte. As program memory is only 24-bits wide, the upper byte from this latter space does not exist, though it is addressable. It is therefore termed the 'phantom' byte.

[0457] For all the table instructions, the calculated EA (using MODE 2 addressing modes) is concatenated with the 8-bit data table page register, TABPAG<7:0>, to form a 23-bit effective programs space address plus a byte select for program memory as shown in

Figure 152. As there are 15-bits of program space address from the calculated EA, the data table page size in program memory is therefore 32K words.

[0458] The LS-bit of the calculated EA becomes the byte select and is used by TBLRDL and TBLWRL (see Program Memory DOS-00204) to select which byte is accessed. The TBLRDL and TBLWRL instructions therefore view program space as byte or aligned word addressable, 16-bit wide, 64K byte pages (i.e. same as data space). EA[0] is ignored for word wide accesses.

[0459] The TBLRDH and TBLWRH instructions are used to access the high order byte of the program address. These instructions also support word or byte access for orthogonality but the high order byte of the program address can only be read from the LS byte as shown in Figure 156. The MS-byte of a TBLRDH word read will always be clear. These instructions therefore also view program space as byte or aligned word addressable, 16-bit wide, 64K byte pages (i.e. same as data space) as shown in Figure 156.

[0460] It is assumed that for most applications that the high byte (P[23:16]) will not be used for data, making the program memory appear 16-bits wide for data storage. It is intended that the high byte contain a illegal opcode trap to protect the device from accidental execution of stored data. The TBLRDH and TBLWRH instructions are primarily provided for array program/verification purposes and for those applications who wish to compress data storage.

HEX Data File Compatibility

[0461] The program space data access described above can be made compatible with HEX format data files by regarding the program memory as 32-bits wide. Inserting

the 'phantom' byte as shown in Figure 157 allows the HEX format byte address to be directly used as the TBLWTL.w and TBLWTH(.b) EA after a single bit right shift.

HEX File compatibility

External Bus Support

[0462] As discussed herein, program space is 24-bits wide which will require either a mix of external FLASH devices to provide all 24-bits in one bus cycle, or several cycles to fetch the 24-bit word in either 8-bit or 16-bit sections. The External Bus Interface (EBI) module will attempt to provide the user maximum flexibility in this area.

[0463] Data access is potentially somewhat simpler as the fundamental data size is 16-bits. To permit single (bus) cycle, 16-bit wide external memory access, the EBI may optionally be configured to read from a 16-bit external bus and then automatically concatenate an 8-bit trap field prior to passing the 24-bit pword to the CPU. A 16-bit external data bus can therefore be provided for data storage without compromising device robustness. The unused portion of the external bus data path can also revert back to I/O.

Clocking Scheme

[0464] Each instruction cycle (T_{cy}) is comprised of four Q cycles (Q1-Q4). These Q clock are derived using simple logic (i.e. there is no requirement to make them non-overlapping) within the core (and each peripheral module) from global QA and QB quadrature clocks. The quadrature clocks are generated by the PLL module. Maintaining minimal skew between QA and QB across the device will be a critical factor in attaining the target performance. The four phase Q cycles provide the timing/designation for the Decode, Read, Process Data, Write etc., of each instruction cycle. Figure 158 shows the

relationship of the Q cycles to the instruction cycle for both MCU and DSP instructions.

The four Q cycles that make up an execution instruction cycle (T_{cy}) can be generalized as:

Q1: Instruction Decode Cycle or forced NOP and source EA calculation

Q2: Source Data Read Cycle or NOP

Q3: Process the Data and destination EA calculation

Q4: Destination Data Write Cycle or NOP

[0465] Each instruction will show the detailed Q cycle operation for the instruction.

Although most instructions follow the scheme above, some issue two reads, others two writes per cycle. From a Q cycle perspective, the DSP instructions differ from in MCU instruction in so much as the DSP instruction can perform two simultaneous source data reads during the Q1/Q2 access from X and Y data space.

Instruction Cycle Timing

[0466] Internally, the program address latch is updated at the start of every Q1, and the instruction is fetched from the program memory and latched into the ROMLATCH using Q4. The PC is actually adjusted (incremented or loaded) during Q4 of the previous cycle but not transferred into the program address latch until the next instruction has started.

[0467] The instruction is decoded and executed during the following Q1 through Q4. The Instruction is decoded during Q1, though some pre-decode of register and addressing mode bit fields during the prior Q4 may be necessary to speed up execution. Care should be taken with any pre-decoding of the instruction to avoid issues (e.g., having to add extra cycles) during interrupt or call returns.

[0468] There are two, independent data space accesses to (possibly) two different addresses during each instruction cycle. During Q1 the (remainder) of the instruction decode is performed and the source operand EA is calculated. During Q2, the source

operand data is fetched from memory or peripherals. The ALU performs the computation during Q3 at the same time as the destination EA is also calculated in one of the AGUs. During Q4 the results are written to the destination location.

[0469] The clocks and instruction execution flow are shown in Figure 159. The data space buses are addressed twice during each cycle with a read (two reads for the DSP instructions) followed by a write. The program space bus is addressed once during each cycle. Note that, due to the longer FLASH access time (around 3 versus 1 Qclk for RAM/registers), program space data reads (table instructions etc.) will present data to the execution unit in Q4. Consequently, these instructions are all 2 cycle operations.

Instruction Flow/Pipelining

[0470] An "Instruction Cycle" consists of four Q cycles (Q1, Q2, Q3, and Q4). The instruction fetch and execute are pipelined such that fetch takes one instruction cycle while decode and execute takes another instruction cycle. However, due to this prefetch mechanism, each instruction effectively executes in one cycle.

Instruction Flow Types

[0471] [0056]—There are 5five types of instruction flows—summarized below—with reference to FIGS. 4A-4E.

[0057] The first type is a normal one1. Normal 1 word one cycle pipelined instruction. These instructions will take one effective cycle to execute as shown by the illustrative example in FIG.in Figure 4A- a.

[0058] The second type is a one2. One word two cycle pipeline flush instruction. These instructions include the relative branches, relative call, skips and returns. When an instruction changes the PC (other than to increment it), the pipelined fetch is discarded. This makes the instruction take two effective cycles to execute as shown in FIG.Figure 4B- b.

[0059] The third type is a table3. Table operation instruction.instructions. These instructions will suspend the fetching to insert a read or write cycle to the program memory. The instruction fetched while executing the table

operation is saved for 1 cycle and executed in the cycle immediately after the table operation as shown in FIG. ~~Figure 4C~~.

[0060] The fourth type is a two ~~4~~. Two word instruction~~instructions~~ for CALL and GOTO. In these instructions, the fetch after the instruction contains the remainder of the jump or call destination addresses. Normally, these instruction~~instructions~~ would require ~~three~~3 cycles to execute, ~~two~~2 for fetching the ~~two~~2 instruction words and ~~one~~1 for the subsequent pipeline flush. However, by providing a high speed path on the second fetch, the PC can be updated with the complete value in the first cycle of instruction execution, resulting in a ~~two~~2 cycle instruction as shown in FIG. ~~Figure 4D~~.

[0061] The fifth type is a two ~~5~~. Two word instruction~~instructions~~ for DO and DOW. In these instructions, the fetch after the instruction contains an address offset. This address offset is added to the first instruction address to generate the last loop instruction address.

Interrupt recognition execution. Instruction cycles during interrupts are shown in the interrupts section.

Program Flow Loop Control

[0472] The dsPIC core supports both REPEAT and DO instruction constructs to provide unconditional automatic program loop control.

[0473] The REPEAT instruction will cause the instruction immediately following to be repeated a fixed number of times as defined by an 14-bit literal encoded in the instruction. The REPEATW instruction will cause the instruction immediately following it to be repeated a fixed number of times as defined by the contents of a W register declared within the instruction, enabling the loop count to be a variable. The loop count is held in the 16-bit RCOUNT register (which is memory mapped) and is thus user accessible. It is initialized by the REPEAT[W] instruction during Q2.

[0474] The instruction to be repeated is prefetched during the REPEAT[W] instruction and held in the ROMLATCH. It is not fetched again for all subsequent iterations, and the Instruction Register is loaded from the locked ROMLATCH.

[0475] For a loop count value equal 1, REPEAT[W] has the effect of a NOP (other than RCOUNT being loaded with 1). The RA (Repeat Active) status bit in the SR is not set during execution of REPEAT[W] and the PC is incremented as would normally be the case during Q4 of an instruction. The repeat loop is essentially disabled before it begins, allowing the next instruction to execute only once while pre-fetching the subsequent instruction (i.e. normal execution flow).

[0476] For loop count values greater than 1, the PC is not incremented as would normally be the case during Q4 of an instruction (and will therefore continue to point to the instruction to be repeated). Further PC increments are inhibited until the loop ends. The RA (Repeat Active) status bit in the SR is also set during execution of REPEAT[W]. See Figure 160 for a functional flow diagram of the REPEAT[W] operation, and Figure 161 for an instruction pipeline example of a REPEAT[W] loop. RA may be a read only bit within the SR and might not be modifiable through software.

[0477] The RCOUNT register is decremented then tested during each instruction iteration. It will equal two at the beginning of the penultimate instruction. The subsequent decrement will make RCOUNT=1, signifying the end of the repeat loop, which causes the RA bit in the SR to be cleared. In addition, the PC increment inhibit is released and the PC bumps in Q4 of this instruction to point to the instruction after the repeated instruction. The last instruction to be repeated is then executed as a normal instruction (i.e. includes an instruction prefetch & PC bump). Testing for the end of loop during the penultimate instruction is required to allow a normal instruction prefetch to occur during the last iteration (i.e. no delays due to 'end of loop' tests).

[0478] A consequence of executing the last instruction outside the repeat loop is that the loop will effectively iterate [loop count +1] times (i.e. a loop count of 0 is not possible). Choosing the loop termination count value to equal one enables the loop count and number of iteration to match for all but RCOUNT equal to zero. For a loop count value of 0, REPEAT will iterate the next instruction 16384 times and REPEATW will iterate the next instruction 65536 times

[0479] The combined instruction flow diagram for REPEAT[W] and DO[W] is shown in Figure 167.

[0480] A REPEAT instruction loop may be interrupted at any time. As is the case for all instructions, the PC update is arranged such that it will not be incremented during the instruction when an exception is acknowledged. For a repeated instruction, the PC update is already inhibited (by the RA bit) which ensures that, upon return, the RETFIE instruction will correctly prefetch said instruction (i.e. the stacked PC will point to the instruction to be repeated).

[0481] Exception processing proceeds as normal, except for a fast interrupt acknowledgment where the contents of the Instruction Latch are transferred into a temp register (IR Temp). This occurs irrespective of the state of the RA bit and is not related to the REPEAT operation. Standard exception processing completes and the ISR is executed as normal in either case.

[0482] Note that, in order to interrupt a REPEAT in progress, the LS-byte of the SR (SRL, which includes the RA bit) is stacked during exception processing. This preserves the state of the RA bit prior to interruption. The RA bit in the SR is then cleared, also during exception processing. In addition, the RCOUNT register has a shadow register

associated with it which is loaded during exception processing (any exception, not just for a fast interrupt). This, in conjunction with the preservation of the RA bit (SRL stacked), permits another REPEAT instruction to be executed within the initial interrupt service routine (i.e. any ISR provided interrupt nesting is not enabled).

[0483] Should interrupt nesting be enabled, subsequent interrupts must stack the RCOUNT register before another REPEAT loop may be executed from within the ISR. If RCOUNT is stacked, the RA preservation feature will also operate for all subsequent nested interrupts. Note that RCOUNT must be restored prior to returning from the ISR. The RA bit is restored automatically during interrupt return processing. Also note that for nested interrupts, the most efficient method to handling REPEAT instructions within ISRs will be to always stack RCOUNT.

[0484] Interrupt return operates as normal and requires no special handling for returning into a REPEAT[W] loop. Normal interrupts will prefetch the repeated instruction during the second cycle of the RETFIE. Return from a fast interrupt will reload the Instruction Latch from the IR Temp register and execute the next repeat iteration during the second cycle. The stacked RA bit will be restored when the SRL register is popped and, if set, the interrupted REPEAT loop will be resumed. Clearing the RA bit in the stacked SR from within an ISR is a method to force an interrupted loop to terminate (subject to one more iteration) after the interrupt returns. RA is not software modifiable within the SR.

[0485] The DO & DOW instructions will execute instructions following the DO[W] until an end address is reached at which time instruction execution will start again at the instruction immediately following the DO[W]. This will be repeated a finite number of

times as defined by either an 14-bit literal encoded in the 1st word of the instruction (for DO) or by the contents of a W register declared within the instruction (for DOW), enabling the loop count to be a variable. The instruction execution order need not be sequential, nor does the loop end address have to be greater than the start address.

[0486] Referring to Figure 166, the DO[W] instruction loads the loop count value into the loop count register (DCOUNT) during Q2. Note that, as it is required that a REPEAT[W] instruction be executable from within a DO loop, the DCOUNT and RCOUNT registers must be independent. The associated decremter can be shared however, the last instruction of a DO[W] loop cannot be:

1. a REPEAT[W] instruction or
2. the instruction within a repeat loop.

Ideally, these circumstances should be detected & flagged by the assembler.

[0487] The loop start address (PC) is stored in the DOSTART register during Q2 of the second cycle. The two cycle DO[W] instruction then calculates the end address by executing a 23-bit signed addition of the current PC[23:1] (which points to the first loop instruction) and a signed 16-bit literal offset encoded within the 2nd word of the DO[W] instruction. This is executed using the MCU ALU during Q1 and Q3 of the 2nd cycle. The loop end address is stored in the DOEND register during Q4. The DOEND and DOSTART registers are closely coupled with the PC as shown in Figure 162. The DA bit within the SR is also set during DO, forcing all subsequent instruction cycles to execute a PC address compare during Q1. This comparison must occur for every cycle (i.e. not just once for a 2 cycle instruction). DO is not required to execute from test memory space. The DOSTART, DOEND registers are therefore restricted to 22-bits each with an additional MS bit always = 0.

[0488] The DO[W] literal address offset is such that the end address is calculated to be the last instruction within the loop. This will cause a valid PC address compare during the Q1 compare operation of the penultimate instruction (i.e. during the prefetch of the last instruction). This will then enable the loop counter to be decremented and tested, and the result combined with the address compare during Q3 of the same instruction.

[0489] If the loop counter after decrement does not equal 1 (as shown in Figure 164), the PC is loaded with the loop start address during Q4 (such that the last instruction will prefetch the first loop instruction, initiating another loop pass). The loop penultimate instruction does not have to be the one immediately preceding the last loop instruction. It can be a branch or GOTO instruction which targets the last instruction as shown in Figure 165 (for a branch).

[0490] If the loop counter after decrement equals 1 (as shown in Figure 166), then the DA bit in the SR is cleared and the PC is incremented as normal during Q4 (such that the last instruction will prefetch the instruction following it and exit the loop).

[0491] The DO loop is equivalent to the 'C' construct DO-WHILE which implies that the loop will be executed at least once. Choosing the loop termination count value to equal one enables the loop count and number of iteration to match for all DCOUNT values except zero.

[0492] For a DCOUNT loop count value of 0, DO will iterate the loop 16384 times and REPEATW will iterate the loop 65536 times. The loop end comparison may be an equality test only. The loop end address must be pre-fetched in order for the end of loop condition to be recognized. That is, exiting the loop to a PC value greater than the end address (or less than the start address) will not cause the loop count to change.

[0493] The combined instruction flow diagram for REPEAT[W] and DO[W] is shown in Figure 167.

[0494] The DOSTART, DOEND and DCOUNT loop registers have a shadow register associated with them which permit a single level of nesting. In addition, as the DOSTART, DOEND and DCOUNT registers are user accessible, they may be manually saved to permit additional nesting. However, it should be noted that the overhead associated with manually saving these registers outweighs the benefits of additional DO loop nesting with the possible exception of a DO loop within an interrupt.

[0495] When a DO is executed, the DOSTART, DOEND and DCOUNT registers are transferred into the shadow registers prior to being updated with the new loop values. The DA bit is also shadowed prior to being set during DO execution. These operations occur for all DO instruction executions, whether nested or not. Similarly, during all loop exits, the shadow contents of the DOSTART, DOEND and DCOUNT registers and the DA bit are transferred back into their respective host registers.

DO Loops and Interrupts

[0496] A DO[W] loop may be interrupted at any time without penalty. Note that, in order to suspend an interrupted DO loop during execution of an ISR, the LS-byte of the SR (SRL, which includes the DA bit) is stacked then cleared (in the SR) during exception processing. Although this is not essential because the DO loop end address is unlikely to be encountered during the ISR, it is consistent with REPEAT operation. If a background DO loop was active (stacked DA bit set), the DOSTART, DOEND and DCOUNT registers must then be stacked before another DO loop may be executed from within the ISR. This applies to any interrupt class. These register must be restored prior to returning from the

ISR. Prior to executing a DO within an interrupt requires stacking and restoring five words of data. This overhead may mean DO is not the most efficient means for loop control within an ISR.

[0497] Interrupt return operates as normal and requires no special handling for returning into a DO[W] loop. The stacked DA bit will be restored into the SRL register and, if set, the interrupted DO loop will resume. Clearing the DA bit in the stacked SR from within an ISR is a method to force an interrupted loop to terminate early after the interrupt returns. The loop will complete the iteration underway and then terminate. If the interrupt occurs during the penultimate or last instruction of the loop, one more iteration of the loop will occur. DA is not software modifiable within the SR.

DO and REPEAT Restrictions

[0498] Any instruction can follow a REPEAT except for:

1. Flow control (any branch, compare and skip, GOTO, CALL, CALLW, RCALL, RETURN or RETLW)
2. Another REPEAT or DO

[0499] As it is not especially useful to execute any of these instructions within a repeat loop, the restrictions on this instruction are minimal.

[0500] REPEAT is interruptible and can be then be nested from within an initial (first, unnested) ISR. If interrupt nesting is enabled, REPEAT can be nested from within any ISR but only after the user stacks the appropriate registers manually (all REPEAT control registers are user accessible).

[0501] All DO loops must contain at least 2 instructions because the loop termination tests are performed in the penultimate instruction. REPEAT should be used for single instruction loops. All other restrictions with regard to the DO loop revolve

around the last instruction. With the notable exception of CALLW, the last instruction should not be:

1. Flow control (any branch, compare and skip, GOTO, RCALL)
2. Another REPEAT or DO
3. Instruction within a repeat loop
4. Any 2 word instruction

[0502] If at all possible, the assembler should be capable of flagging these instructions if placed at the end of a DO loop.

[0503] The (one word) CALLW will function correctly at the end of a DO loop because the stacked PC will address the start of loop instruction (to fetch upon return).

[0504] PC relative instructions (e.g. RCALL, branches) won't work correctly at the end of a loop because the PC calculation will be performed using the current PC value which will be the loop start address. That is, the assembler psuedo-PC and the real PC do not match at this point.

[0505] Should execution of a REPEAT[W] instruction as the last loop instruction be attempted, the DO[W] loop counter will take priority and the REPEAT target instruction will never be executed before the DO[W] loop jumps to the loop start. Should the last loop instruction be the instruction being repeated within a REPEAT loop, the DO[W] loop counter will also take priority and the REPEAT target instruction will only execute once with no change to RCOUNT before the DO[W] loop jumps to the loop start.

[0506] Two-word instructions will fail if placed at the end of a DO loop because the PC is adjusted in the penultimate instruction in order to accommodate the instruction prefetch (without a dead cycle). Consequently, the second word of a two-word instruction would therefore be incorrectly fetched from the loop start address.

[0507] RETURN and RETLW will work correctly when the last instruction of a DO loop but the user must be responsible for returning into the loop to complete it.

~~[0062]~~ ~~Programmers~~ Programmer Model

[0508] ~~[0063]~~ The programmers model of the processor is shown in FIG. ~~Figure~~ Figure 5 and consists of ~~16.times.16~~ 16 ~~x~~ 16-bit working registers, ~~2.times.2~~ 2 ~~x~~ 40-bit accumulators, status register, data table page register, data space program page register, DO and REPEAT registers, and program counter. The working registers can act as data, address or offset registers. All registers are memory mapped.

[0509] ~~[0064]~~ Most of these registers have a shadow register associated with them as shown in FIGS. 1-33. Figure 5. The shadow register is used as a temporary holding register and can transfer its contents to or from its host register upon some event occurring. None of the shadow registers are accessible directly. The following rules apply to register transfer into and out of shadows.

- ~~[0065]~~ Fast Interrupts entry & exit ~~[0066]~~
 - W0 to W14 shadows transferred ~~[0067]~~
 - PC shadow transferred ~~[0068]~~
 - TABPAG & DSPPAG shadows transferred ~~[0069]~~
 - RCOUNT shadow transferred ~~[0070]~~
 - SR[6:0] shadow bits transferred
- ~~[0071]~~ Normal Interrupt Entry ~~[0072]~~
 - RCOUNT shadow transferred ~~[0073]~~
 - SR[6] shadow bit transferred
- ~~[0074]~~ Nested DO ~~[0075]~~
 - DOSTART, DOEND, DCOUNT shadows loaded

[0510] ~~[0076]~~ Byte instructions which target the working register array only effect the least significant byte of the target register. However, a consequence of memory mapped working

registers is that both the least and most significant bytes can be manipulated through byte wide data memory space accesses.

~~{0077}~~ Uninitialized W Register Trap

[0511] ~~{0078}~~ The W register array (except W15) is not effected by a reset and therefore must be considered uninitialized until a written to. An attempt to read an uninitialized register for an address access will generate an address error trap (fetch of an uninitialized address). In this situation, the user will most likely choose to reset the application, though recovery may be possible through an examination of the problematic instruction (via the stacked return address).

[0512] ~~{0079}~~ This function is achieved through the addition of a single latch to each W register (W0 through W14). The latch is cleared by reset and set by the first write to the associated register and is described in the patent application entitled "Register Point, as shown in [See Uninitialized W Register Trap]" incorporated by reference herein. When the latch is clear, a read of the corresponding register to either AGU will force an address error trap. W15 is initialized during reset (see [See Software Stack Pointer]) and consequently does not require this feature.

~~{0080}~~ Default W Register Selection

[0513] ~~{0081}~~ The default W register for all file register instructions is defined by the WD[3:0] field in the CORCON (~~CORE CONTROL~~ Core Control register). This field is reset to 0x0000, corresponding to register W0. ~~As most of the CORCON function relates to DSP operations, it is discussed in Section 2.0, DSP Engine.~~

~~{0082}~~ Software Stack Pointer

[0514] ~~{0083}~~ W15 has been dedicated as the software stack pointer, and will be automatically modified by exception processing and subroutine calls and returns. However, W15 can be referenced by any instruction in the same manner as all other W registers. This

simplifies reading, writing and manipulating the stack pointer (e.g. creating stack frames). In order to protect against misaligned stack accesses, W15[0] ~~may be clear~~ is always clear.

[0515] ~~{0084}~~ W15 ~~may be~~ is initialized to 0x0200 during a reset. This will point to valid RAM in all derivatives and will guarantee stack availability for non-maskable trap exceptions or priority level 7 interrupts which may occur before the SP is set to where the user desires it. The user may reprogram the SP during initialization to any location within data space.

[0516] ~~{0085}~~ W14 ~~may be~~ has been dedicated as a stack frame pointer as defined by the LNK and ULNK instructions. However, W14 can be referenced by any instruction in the same manner as all other W registers.

[0517] ~~{0086}~~ The stack pointer always points to the first available free word and fills working from lower towards higher addresses. It pre-decrements for stack pops (reads) and post increments for stack ~~pushes~~ pushes (writes) as shown in FIGS. 1-32. Figure 172. Note that for a PC push during any CALL instruction, the MS-byte of the PC is zero extended before the push, ensuring that the MS-byte is always clear. The stack timing is shown in FIGS. 1-34. Figures 170 and 171. A PC push during exception processing ~~may~~ will concatenate the SRL register to the MS-byte of the PC prior to the push.

~~{0087} Stack Pointer Overflow Trap~~

[0518] ~~{0088}~~ There is a stack limit register (SPLIM) associated with the stack pointer that is uninitialized at reset. SPLIM[15:1] is a 15-bit register. As is the case for the stack pointer, SPLIM[0] is forced to 0 because all stack operations must be word aligned.

[0519] ~~{0089}~~ The stack overflow check ~~may~~ will not be enabled until a word write to SPLIM occurs after which time it can only be disabled by a reset. All EA's generated using W15 as Wsrc or Wdst (but not Wb) are compared against the value in SPLIM. Should the EA be greater than the contents of SPLIM, then a stack error trap is generated. This comparison is a

subtraction, so the trap will occur for any SP greater than SPLIM. In addition, should the SP EA calculation wrap over the end of data space (0xFFFF), AGU X will generate a carry signal which will also cause a stack error trap (if the SPLIM register has been initialized).

~~{0090} Stack Pointer Underflow Trap~~

[0520] ~~{0091}~~ The stack is initialized to 0x0200 during reset. A simple stack underflow mechanism is provided which will initiate a stack error trap should the stack pointer address ever be less than 0x0200.

~~{0092} Status Register~~

[0521] ~~{0093}~~ The ~~status register is~~ dsPIC core has a 16-bit status register (SR), the LS-byte of which is referred to as the lower status register (SRL). ~~A detailed table showing the arrangement of the SR register is set forth below.~~

~~4 Upper Half: R/W-0 R/W-0 R/W-0 R/W-0 R/W-0 R/W-0 U U OA OB SA SB OAB SAB — bit 15 bit 8 Lower Half: R-0 R-0 R/W-0 R/W-0 R/W-0 R/W-0 R/W-0 R/W-0 DA RA SZ N OV Z DC C bit 7 bit 0~~ [0094] The SRL contains A detailed description is shown in Table 187. SRL contains all the MCU ALU operation status flags (including ~~at~~ the new 'sticky Z' (SZ) bit described in the application entitled "Sticky Zero Bit Flag" incorporated by reference herein and' (SZ) bit) plus the REPEAT and DO loop active status bits. During exception processing, SRL may be ~~is~~ concatenated with the MS-byte of the PC to form a complete word value which is then stacked. The upper byte of the SR contains the DSP Adder/Subtractor status bits.

[0522] ~~{0095}~~ The upper byte of the SR ~~may contains the DSP Adder/Subtractor status bits.~~ All SR bits are read/write except for the DA and RA bits which are read only because accidentally setting them could cause erroneous operation (include inhibiting PC increments). When the memory mapped SR is the destination address for an operation which affects the any of the SR bits, data writes are disabled to all bits. ~~The bits of the SR are summarized below.~~

~~5 bit 15 OA: Accumulator A Overflow Status 1= Accumulator A overflowed 0= Accumulator A not overflowed bit 14 OB: Accumulator B Overflow Status 1= Accumulator B overflowed 0= Accumulator B not overflowed bit 13 SA: Accumulator A Saturation 'Sticky' Status 1= Accumulator A is saturated or has been saturated at some time 0= Accumulator A is not saturated bit 12 SB: Accumulator B Saturation 'Sticky' Status 1= Accumulator B is saturated or has been saturated at some time 0= Accumulator B is not saturated bit 11 OAB: OA OB~~

Combined Accumulator Overflow Status 1= Accumulators A or B have overflowed 0= Neither Accumulators A or B have overflowed bit 10 SAB: SA SB Combined Accumulator 'Sticky' Status 1= Accumulators A or B are saturated or have been saturated at some time in the past 0= Neither Accumulator A or B are saturated bit 9-8 Unused bit 7 DA: DO Loop Active 1= DO loop in progress 0= DO loop not in progress bit 6 RA: REPEAT Loop Active 1= REPEAT loop in progress 0= REPEAT loop not in progress bit 5 SZ: MC ALU 'sticky Zero bit 1= An operation which effects the Z bit has set it at some time in the past 0= The most recent operation which effects the Z bit has cleared it (i.e. a non-zero result) bit 4 N: MCU ALU Negative bit bit 3 OV: MCU ALU Overflow bit bit 2 Z: MCU ALU Zero bit bit 1 DC: MCU ALU Half Carry/Borrow bit bit 0 C: MCU ALU Carry/Borrow bit Legend R = Readable bit W = Writable bit U = Unimplemented bit, read as '0' n = Value at POR 1 = bit is set 0 = bit is cleared x = bit is unknown

[0096] Instruction Addressing Modes

[0097] The basic set of addressing modes shown in Table 4-1. Note that, 'Wn+' indicates that the contents of Wn is added to something to form the effective address which is then written back into Wn. 'Wn+' indicates that the contents of Wn is added to something to form the effective address but the contents of Wn remain unchanged.

[0098] The addressing modes in form the basis of three groups of addressing modes optimized to support specific instruction features. They are MODE1, MODE2 AND MODE3. The DSP MAC and derivative instructions are an exception where the addressing modes are encoded differently. This set of addressing modes is referred to as MODE4.

6 Note: Reference DSP CORE DOS FOR MODE4 Addressing Mode Function Description Register Direct EA = Wn Wn is the EA Register Indirect EA = [Wn] The content of Wn forms the EA Register Indirect Post EA = [Wn] + 1 The contents of Wn forms the EA modified which is post-modified by a constant value Register Indirect Pre-modified EA = [Wn + 1] Wn is pre-modified by a signed EA = [Wn - 1] constant value to form the EA Register Indirect with Register EA = [Wn + Wb] The sum of Wn and Wb forms the EA Offset Register Indirect with Constant EA = [Wn + The sum of Wn and a signed constant Offset constant] value forms the EA

[0099] EA is defined as the effective address. All address modification values (except Wb) are scaled for word access.

[0100] Addressing Modes

[0101] All but few instructions support both 8-bit and 16-bit operand data sizes. In order to efficiently accommodate this requirement, effective addresses are byte-aligned. As the data space is 16-bits wide, the following consequences must be understood.

[0102] a. Mis-aligned word accesses are not supported. All word effective addresses must be even (the LS-bit of the EA is ignored by the data space memory).

[0103] b. The LS-bit of the effective address is used to select which byte (upper or lower) is multiplexed onto bits [7:0] of the data bus for byte sized accesses.

[0104] e. Post and pre-modification of a register by a constant value to create a new effective address must take into account of the data size accessed. All constant values, whether implied (e.g. post inc) or declared (e.g. post-modify with S5lit) are scaled by a factor of 2 for word accesses. For example:

[0105] $[Ws] += 1$ will post modify data source pointer Ws by 1 for a byte access, and by 2 for a word access.

[0106] $[Ws] += Slit5$ will post modify data source pointer Ws by Slit5 for byte accesses and $Slit5 << 1$ (shift left by 1) for word accesses.

[0107] Address modification values (except Wb) are scaled for word access.

TABLE 187 -- SR, CPU Status Register (0xXXXX)

Upper Half:

<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>U</u>	<u>U</u>
<u>OA</u>	<u>OB</u>	<u>SA</u>	<u>SB</u>	<u>OAB</u>	<u>SAB</u>	<u>=</u>	<u>=</u>
<u>bit 15</u>							<u>bit 8</u>

Lower Half:

<u>R-0</u>	<u>R-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>	<u>R/W-0</u>
<u>DA</u>	<u>RA</u>	<u>SZ</u>	<u>N</u>	<u>OV</u>	<u>Z</u>	<u>DC</u>	<u>C</u>
<u>bit 7</u>							<u>bit 0</u>

OA: Accumulator A Overflow Status

15 1 = Accumulator A overflowed

0 = Accumulator A not overflowed

OB: Accumulator B Overflow Status

14 1 = Accumulator B overflowed

0 = Accumulator B not overflowed

SA: Accumulator A Saturation 'Sticky' Status

13 1 = Accumulator A is saturated or has been saturated at some time

0 = Accumulator A is not saturated

SB: Accumulator B Saturation 'Sticky' Status

12 1 = Accumulator B is saturated or has been saturated at some time

0 = Accumulator B is not saturated

OAB: OA || OB Combined Accumulator Overflow Status

11 1 = Accumulators A or B have overflowed

0 = Neither Accumulators A or B have overflowed

SAB: SA || SB Combined Accumulator 'Sticky' Status

10 1 = Accumulators A or B are saturated or have been saturated at some time in the past

0 = Neither Accumulator A or B are saturated

9-8 Unused

DA: DO Loop Active

7 1 = DO loop in progress

0 = DO loop not in progress

RA: REPEAT Loop Active

6 1 = REPEAT loop in progress

0 = REPEAT loop not in progress

SZ: MCU ALU 'sticky' Zero bit

5 1 = An operation which effects the Z bit has set it at some time in the past

0 = The most recent operation which effects the Z bit has cleared it (i.e. a non-zero result)

4 N: MCU ALU Negative bit

3 OV: MCU ALU Overflow bit

2 Z: MCU ALU Zero bit

1 DC: MCU ALU Half Carry/ Borrow bit

0 C: MCU ALU Carry/ Borrow bit

Legend

R = Readable bit

W = Writable bit

U = Unimplemented bit, read as '0'

-n = Value at POR

1 = bit is set

0 = bit is cleared

x = bit is unknown

Exceptions and Stack

[0523] The core supports a prioritized interrupt and trap exception scheme. There are up to eight levels of interrupt priority, each of which has an interrupt vector associated with it. Each interrupt source is user programmable with regard to what priority (and therefore vector address) it uses. The highest priority interrupt is non-maskable. There are seven traps available to improve operational robustness, all of which are non-maskable. They adhere to a predefined priority scheme.

[0524] Stacking associated with exceptions and subroutine calls is executed on a software stack. Register W15 is dedicated as the stack pointer and has the LSB = 0.

TABLE 188 -- Central Processing Unit Status Flag Operations

<u>PLA</u> <u>Mnemonic</u>	<u>Status Affected</u>	<u>C</u>	<u>DC</u>	<u>N</u>	<u>OV</u>	<u>SZ</u>	<u>Z</u>	<u>OA</u>	<u>OB</u>	<u>SA</u>	<u>SB</u>
Move Operations											
<u>EXCH</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
<u>LDDW</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
<u>LDQW</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
<u>LDW</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
<u>MOV</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
<u>MOVE</u>	<u>N,Z</u>	==	==		==	==		==	==	==	==
<u>MOVL</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
<u>MOVLW</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
<u>MOVWF</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
<u>STDW</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
<u>STQW</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
<u>STW</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
Table Operations											
<u>TBLRDH</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
<u>TBLRDL</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
<u>TBLWTH</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
<u>TBLWTL</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
Math Operations - W Registers											
<u>ADD</u>	<u>C,DC,N,OV,SZ,Z</u>							==	==		
<u>ADDC</u>	<u>C,DC,N,OV,SZ,Z</u>							==	==	==	==
<u>AND</u>	<u>N,SZ,Z</u>	==	==		==			==	==	==	==
<u>IOR</u>	<u>N,SZ,Z</u>	==	==		==			==	==	==	==
<u>SUB</u>	<u>C,DC,N,OV,SZ,Z</u>							==	==	==	==
<u>SUBB</u>	<u>C,DC,N,OV,SZ,Z</u>							==	==	==	==
<u>SUBR</u>	<u>C,DC,N,OV,SZ,Z</u>							==	==	==	==
<u>SUBBR</u>	<u>C,DC,N,OV,SZ,Z</u>							==	==	==	==
<u>XOR</u>	<u>N,SZ,Z</u>	==	==		==			==	==	==	==
Math Operations - Short Literals (literal 0..31)											
<u>ADDLS</u>	<u>C,DC,N,OV,SZ,Z</u>							==	==	==	==
<u>ADDCLS</u>	<u>C,DC,N,OV,SZ,Z</u>							==	==	==	==
<u>ANDLS</u>	<u>N,SZ,Z</u>	==	==		==			==	==	==	==
<u>IORLS</u>	<u>N,SZ,Z</u>	==	==		==			==	==	==	==
<u>SUBLS</u>	<u>C,DC,N,OV,SZ,Z</u>							==	==	==	==
<u>SUBBLS</u>	<u>C,DC,N,OV,SZ,Z</u>							==	==	==	==
<u>SUBRLS</u>	<u>C,DC,N,OV,SZ,Z</u>							==	==	==	==
<u>SUBBRLS</u>	<u>C,DC,N,OV,SZ,Z</u>							==	==	==	==
<u>XORLS</u>	<u>N,SZ,Z</u>	==	==		==			==	==	==	==
Math Operations - W Registers Single Operand											
<u>CLR</u>	<u>Z</u>	==	==	==	==	==	1	==	==	==	==
<u>COM</u>	<u>N,SZ,Z</u>	==	==		==			==	==	==	==

<u>DEC</u>	<u>C,DC,N,OV,SZ,Z</u>								==	==	==	==
<u>DEC2</u>	<u>C,DC,N,OV,SZ,Z</u>								==	==	==	==
<u>INC</u>	<u>C,DC,N,OV,SZ,Z</u>								==	==	==	==
<u>INC2</u>	<u>C,DC,N,OV,SZ,Z</u>								==	==	==	==
<u>NEG</u>	<u>C,DC,N,OV,SZ,Z</u>								==	==	==	==
<u>SETM</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
Math Operations - File Registers												
<u>ADDWF</u>	<u>C,DC,N,OV,SZ,Z</u>								==	==	==	==
<u>ADDWFC</u>	<u>C,DC,N,OV,SZ,Z</u>								==	==	==	==
<u>ANDWF</u>	<u>N,SZ,Z</u>	==	==		==				==	==	==	==
<u>IORWF</u>	<u>N,SZ,Z</u>	==	==		==				==	==	==	==
<u>SUBFW</u>	<u>C,DC,N,OV,SZ,Z</u>								==	==	==	==
<u>SUBBFW</u>	<u>C,DC,N,OV,SZ,Z</u>								==	==	==	==
<u>SUBWF</u>	<u>C,DC,N,OV,SZ,Z</u>								==	==	==	==
<u>SUBBWF</u>	<u>C,DC,N,OV,SZ,Z</u>								==	==	==	==
<u>XORWF</u>	<u>N,SZ,Z</u>	==	==		==				==	==	==	==
Math Operations - File Registers Single Operand												
<u>CLRF</u>	<u>Z</u>	==	==	==	==	==	1		==	==	==	==
<u>COMF</u>	<u>N,SZ,Z</u>	==	==		==				==	==	==	==
<u>DECF</u>	<u>C,DC,N,OV,SZ,Z</u>								==	==	==	==
<u>INCF</u>	<u>C,DC,N,OV,SZ,Z</u>								==	==	==	==
<u>NEGF</u>	<u>C,DC,N,OV,SZ,Z</u>								==	==	==	==
<u>SETF</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
Math Operations - Literals (literal -512..511)												
<u>ADDLW</u>	<u>C,DC,N,OV,SZ,Z</u>								==	==	==	==
<u>ADDCLW</u>	<u>C,DC,N,OV,SZ,Z</u>								==	==	==	==
<u>ANDLW</u>	<u>N,SZ,Z</u>	==	==		==				==	==	==	==
<u>IORLW</u>	<u>N,SZ,Z</u>	==	==		==				==	==	==	==
<u>SUBLW</u>	<u>C,DC,N,OV,SZ,Z</u>								==	==	==	==
<u>SUBBLW</u>	<u>C,DC,N,OV,SZ,Z</u>								==	==	==	==
<u>XORLW</u>	<u>N,Z</u>	==	==		==				==	==	==	==
Math Operations - Multiply, Adjust												
<u>DAW</u>	<u>C</u>		==	==	==	==	==	==	==	==	==	==
<u>DIV</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>MULS</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>MULSU</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>MULSULS</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>MULU</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>MULULS</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>MULUS</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>MULWF</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>SE</u>	<u>C,N,Z</u>		==		==		==		==	==	==	==
<u>ZE</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>SWAP</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
Rotate/Shift Operations - W Registers												

<u>ASR</u>	<u>C,N,OV,SZ,Z</u>		==					==	==	==	==
<u>LSR</u>	<u>C,N,OV,SZ,Z</u>		==					==	==	==	==
<u>RLC</u>	<u>C,N,SZ,Z</u>		==		==			==	==	==	==
<u>RLNC</u>	<u>N,SZ,Z</u>	==	==		==			==	==	==	==
<u>RRC</u>	<u>C,N,SZ,Z</u>		==		==			==	==	==	==
<u>RRNC</u>	<u>N,SZ,Z</u>	==	==		==			==	==	==	==
<u>SL</u>	<u>C,N,OV,SZ,Z</u>		==					==	==	==	==
Rotate/Shift Operations - File Registers											
<u>ASRF</u>	<u>C,N,OV,SZ,Z</u>		==					==	==	==	==
<u>LSRF</u>	<u>C,N,OV,SZ,Z</u>		==					==	==	==	==
<u>RLCF</u>	<u>C,N,SZ,Z</u>		==		==			==	==	==	==
<u>RLNCF</u>	<u>N,SZ,Z</u>	==	==		==			==	==	==	==
<u>RRCF</u>	<u>C,N,SZ,Z</u>		==		==			==	==	==	==
<u>RRNCF</u>	<u>N,SZ,Z</u>	==	==		==			==	==	==	==
<u>SLF</u>	<u>C,N,OV,SZ,Z</u>		==					==	==	==	==
Barrel Shift Operations - W Registers (shift range -16..15)											
<u>ASRW</u>	<u>C,SZ,Z</u>		==	==	==			==	==	==	==
<u>LSRW</u>	<u>C,SZ,Z</u>		==	==	==			==	==	==	==
<u>MSLW</u>	<u>C,SZ,Z</u>		==	==	==			==	==	==	==
<u>MSRW</u>	<u>C,SZ,Z</u>		==	==	==			==	==	==	==
<u>SLW</u>	<u>C,SZ,Z</u>		==	==	==			==	==	==	==
Barrel Shift Operations - Short Literals (shift range -16..15)											
<u>ASRK</u>	<u>C,SZ,Z</u>		==	==	==			==	==	==	==
<u>LSRK</u>	<u>C,SZ,Z</u>		==	==	==			==	==	==	==
<u>MSLK</u>	<u>C,SZ,Z</u>		==	==	==			==	==	==	==
<u>MSRK</u>	<u>C,SZ,Z</u>		==	==	==			==	==	==	==
<u>SLK</u>	<u>C,SZ,Z</u>		==	==	==			==	==	==	==
DSP OPERATIONS - Accumulator Ops											
<u>ADDAB</u>	<u>OA,OB,SA,SB</u>	==	==	==	==	==	==				
<u>ADDAC</u>	<u>OA,OB,SA,SB</u>	==	==	==	==	==	==				
<u>LAC</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
<u>NEGAB</u>	<u>OA,OB,SA,SB</u>	==	==	==	==	==	==				
<u>SAC</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
<u>SFTAC</u>	<u>OA,OB,SA,SB</u>	==	==	==	==	==	==				
<u>SFTACK</u>	<u>OA,OB,SA,SB</u>	==	==	==	==	==	==				
<u>SRAC</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
<u>SUBAB</u>	<u>OA,OB,SA,SB</u>	==	==	==	==	==	==				
DSP OPERATIONS - MAC Ops											
<u>CLRAC</u>	<u>OA,OB,SA,SB</u>	==	==	==	==	==	==				
<u>ED</u>	<u>OA,OB,SA,SB</u>	==	==	==	==	==	==				
<u>EDAC</u>	<u>OA,OB,SA,SB</u>	==	==	==	==	==	==				
<u>MAC</u>	<u>OA,OB,SA,SB</u>	==	==	==	==	==	==				
<u>MOVSAC</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
<u>MPY</u>	<u>OA,OB,SA,SB</u>	==	==	==	==	==	==				
<u>MPYN</u>	<u>OA,OB,SA,SB</u>	==	==	==	==	==	==				

MSC	OA,OB,SA,SB	==	==	==	==	==	==				
SQR	OA,OB,SA,SB	==	==	==	==	==	==				
SQRAC	OA,OB,SA,SB	==	==	==	==	==	==				
BIT OPERATIONS - W Registers											
BCLR	None	==	==	==	==	==	==	==	==	==	==
BSET	None	==	==	==	==	==	==	==	==	==	==
BSW	None	==	==	==	==	==	==	==	==	==	==
BTG	None	==	==	==	==	==	==	==	==	==	==
BTST	C or Z		==	==	==	==		==	==	==	==
BTSTS	C or Z		==	==	==	==		==	==	==	==
BTSTW	C or Z		==	==	==	==		==	==	==	==
BIT OPERATIONS - File Registers											
BCLRF	None	==	==	==	==	==	==	==	==	==	==
BSETF	None	==	==	==	==	==	==	==	==	==	==
BTGF	None	==	==	==	==	==	==	==	==	==	==
BTSTF	Z	==	==	==	==	==		==	==	==	==
BTSTSF	Z	==	==	==	==	==		==	==	==	==
BIT FIND OPERATIONS											
FBCL	SZ,Z	==	==	==	==			==	==	==	==
FBCR	SZ,Z	==	==	==	==			==	==	==	==
FF0L	SZ,Z	==	==	==	==			==	==	==	==
FF0R	SZ,Z	==	==	==	==			==	==	==	==
FF1L	SZ,Z	==	==	==	==			==	==	==	==
FF1R	SZ,Z	==	==	==	==			==	==	==	==
Skip OPERATIONS - W Registers											
BTSC	None	==	==	==	==	==	==	==	==	==	==
BTSS	None	==	==	==	==	==	==	==	==	==	==
Skip OPERATIONS - File Registers											
BTFS	None	==	==	==	==	==	==	==	==	==	==
BTFSF	None	==	==	==	==	==	==	==	==	==	==
Inc/Dec Skip OPERATIONS - File Registers											
DECFSNZ	None	==	==	==	==	==	==	==	==	==	==
DECFSZ	None	==	==	==	==	==	==	==	==	==	==
INCFSNZ	None	==	==	==	==	==	==	==	==	==	==
INCFSZ	None	==	==	==	==	==	==	==	==	==	==
Compare OPERATIONS - W Registers											
CP0	C,DC,N,OV,SZ,Z							==	==	==	==
CP1	C,DC,N,OV,SZ,Z							==	==	==	==
CP	C,DC,N,OV,SZ,Z							==	==	==	==
CPB	C,DC,N,OV,SZ,Z							==	==	==	==
Compare OPERATIONS - Short Literals (literal 0...31)											
CPLS	C,DC,N,OV,SZ,Z							==	==	==	==
CPBLS	C,DC,N,OV,SZ,Z							==	==	==	==
Compare OPERATIONS - File Registers											
CPF0	C,DC,N,OV,SZ,Z							==	==	==	==

<u>CPF1</u>	<u>C,DC,N,OV,SZ,Z</u>								==	==	==	==
<u>CPF</u>	<u>C,DC,N,OV,SZ,Z</u>								==	==	==	==
<u>CPFB</u>	<u>C,DC,N,OV,SZ,Z</u>								==	==	==	==
Compare Skip OPERATIONS - File Registers												
<u>CPFSEQ</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>CPFSGT</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>CPFSLT</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>CPFSNE</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
Branch Operations												
<u>BC</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>BGE</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>BGT</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>BGTU</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>BLE</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>BLEU</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>BLT</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>BN</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>BNC</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>BNN</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>BNOV</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>BNZ</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>BOA</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>BOB</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>BOV</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>BRA</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>BSA</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>BSB</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>BZ</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
Jump/Call/Return Operations												
<u>BRAW</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>CALL</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>CALLW</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>GOTO</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>GOTOW</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>RCALL</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>RCALLW</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>RETFIE</u>	<u>INTLV</u>	==	==	==	==	==	==	==	==	==	==	==
<u>RETLW</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>RETURN</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>TRAP</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
Looping Operations												
<u>DO</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>DOW</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>REPEAT</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==
<u>REPEATW</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==	==

<u>Stack Operations</u>											
<u>ITCH</u>	<u>All</u>										
<u>LNK</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
<u>POP</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
<u>PUSH</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
<u>SCRATCH</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
<u>ULNK</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
<u>Control Operations</u>											
<u>CLRWDI</u>	<u>TO,PD</u>	==	==	==	==	==	==	==	==	==	==
<u>DISI</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
<u>HALT</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
<u>NOP</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
<u>NOPR</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
<u>RESET</u>	<u>None</u>	==	==	==	==	==	==	==	==	==	==
<u>SLEEP</u>	<u>TO,PD</u>	==	==	==	==	==	==	==	==	==	==

TABLE 189 -- Central Processing Unit Opcode Field Descriptions

<u>Field</u>	<u>Description</u>
<u>A</u>	<u>Accumulator selection bit: 0=ACCA; 1=ACCB</u>
<u>B</u>	<u>Byte mode selection bit: 0=word operation; 1=byte operation</u>
<u>D</u>	<u>Destination address bit: 0=result stored in Wd; 1=result stored in file register</u>
<u>S</u>	<u>Push or Pop shadows: 0=no shadows; 1=use shadows</u>
<u>z</u>	<u>Bit test destination: 0=C flag bit; 1=Z flag bit</u>
<u>dddd</u>	<u>Wd destination register select: 0000=W0; 1111=W15</u>
<u>ssss</u>	<u>Ws source register select: 0000=W0; 1111=W15</u>
<u>www</u>	<u>Wb base register select: 0000=W0; 1111=W15</u>
<u>ppp</u>	<u>Addressing mode for Ws source register (See Table 190)</u>
<u>qqq</u>	<u>Addressing mode for Wd destination register (See Table 191)</u>
<u>ggg</u>	<u>Literal offset addressing mode for Ws source register (See Table 192)</u>
<u>hhh</u>	<u>Literal offset addressing mode for Wd destination register (See Table 193)</u>
<u>xx</u>	<u>Pre-fetch X Destination (See Table 195)</u>
<u>yy</u>	<u>Pre-fetch Y Destination (See Table 198)</u>
<u>iii</u>	<u>Pre-fetch X Operation (See Table 194)</u>
<u>iii</u>	<u>Pre-fetch Y Operation (See Table 197)</u>
<u>mmm</u>	<u>Multiplier source select (See Table 196)</u>
<u>aa</u>	<u>Accumulator write back mode (See Table 199)</u>
<u>rrrr</u>	<u>Barrel shift count</u>
<u>bbb</u>	<u>3-bit bit position select: 000=LSB; 111=Mr. Soliman Boustany</u>
<u>bbbb</u>	<u>4-bit bit position select: 0000=LSB; 1111=Mr. Soliman Boustany</u>
<u>f ffff ffff ffff</u>	<u>13-bit register file address (0x0000 to 0x1FFF)</u>
<u>ffff ffff ffff ffff</u>	<u>16-bit register file address (0x0000 to 0xFFFF)</u>
<u>k kkkk</u>	<u>5-bit literal field, constant data or label</u>
<u>kkkk kkkk</u>	<u>8-bit literal field, constant data or label</u>
<u>kk kkkk kkkk</u>	<u>10-bit literal field, constant data or label</u>
<u>kk kkkk kkkk kkkk</u>	<u>14-bit literal field, constant data or label</u>
<u>kkkk kkkk kkkk kkkk</u>	<u>16-bit literal field, constant data or label</u>
<u>n</u>	<u>1-bit vector select for trap instructions</u>
<u>nnnn nnnn nnnn</u>	<u>16-bit program offset field for relative branch/call instructions</u>
<u>nnnn</u>	
<u>nnnn nnnn nnnn</u>	<u>23-bit program address for goto/call instructions</u>
<u>nnn0 nnn nnnn</u>	
<u>xxxx xxxx xxxx xxxx</u>	<u>16-bit unused field (don't care)</u>

TABLE 190 -- Addressing Modes For Ws Source Register (Address Mode 1)

<u>ppp</u>	<u>Addressing Mode</u>	<u>Source Operand</u>	<u>Instruction Operation (3)</u>	<u>Effective Address</u>
<u>000</u>	<u>Register Direct</u>	<u>Ws</u>	<u>Wd = Ws op Wb</u>	<u>EAs = W register number</u>
<u>001</u>	<u>Indirect</u>	<u>[Ws]</u>	<u>Wd = [Ws] op Wb</u>	<u>EAs = Ws</u>
				<u>EAs = Ws;</u>
<u>010</u>	<u>Indirect with post-decrement</u>	<u>[Ws]--</u>	<u>Wd = [Ws]-- op Wb</u>	<u>Ws <- (Ws - 1) ⁽¹⁾</u>
				<u>- or -</u>
				<u>Ws <- (Ws - 2) ⁽²⁾</u>
				<u>EAs = Ws;</u>
<u>011</u>	<u>Indirect with post-increment</u>	<u>[Ws]++</u>	<u>Wd = [Ws]++ op Wb</u>	<u>Ws <- (Ws + 1) ⁽¹⁾</u>
				<u>- or -</u>
				<u>Ws <- (Ws + 2) ⁽²⁾</u>
				<u>Ws <- (Ws - 1) ⁽¹⁾ ;</u>
<u>100</u>	<u>Indirect with pre-decrement</u>	<u>[Ws--]</u>	<u>Wd = [Ws--] op Wb</u>	<u>- or -</u>
				<u>Ws <- (Ws - 2) ⁽²⁾ ;</u>
				<u>EAs = Ws</u>
				<u>Ws <- (Ws + 1) ⁽¹⁾ ;</u>
<u>101</u>	<u>Indirect with pre-increment</u>	<u>[Ws++]</u>	<u>Wd = [Ws++] op Wb</u>	<u>- or -</u>
				<u>Ws <- (Ws + 2) ⁽²⁾ ;</u>
				<u>EAs = Ws</u>
<u>11k</u>	<u>(Specifies Slit5 Source for Short Literal Instructions)</u>			
<u>Note:</u>	<u>1: For byte operations, add or subtract 1.</u>			
	<u>2: For word operations, add or subtract 2.</u>			
	<u>3: Wd assumed to be in register direct mode (qqq=000).</u>			

TABLE 191 -- Addressing Modes For Wd Destination Register (Address Mode 2)

<u>qqq</u>	<u>Addressing Mode</u>	<u>Destination Operand</u>	<u>Instruction Operation (3)</u>	<u>Effective Address</u>
<u>000</u>	<u>Register Direct</u>	<u>Wd</u>	<u>Wd = Ws op Wb</u>	<u>EAd = W register number</u>
<u>001</u>	<u>Indirect</u>	<u>[Wd]</u>	<u>[Wd] = Ws op Wb</u>	<u>EAd = Wd</u> <u>EAd = Wd;</u>
<u>010</u>	<u>Indirect with post-decrement</u>	<u>[Wd]--</u>	<u>[Wd]-- = Ws op Wb</u>	<u>Wd <- (Wd - 1) ⁽¹⁾</u> <u>- or -</u> <u>Wd <- (Wd - 2) ⁽²⁾</u> <u>EAd = Wd;</u>
<u>011</u>	<u>Indirect with post-increment</u>	<u>[Wd]++</u>	<u>[Wd]++ = Ws op Wb</u>	<u>Wd <- (Wd + 1) ⁽¹⁾</u> <u>- or -</u> <u>Wd <- (Wd + 2) ⁽²⁾</u> <u>Wd <- (Wd - 1) ⁽¹⁾ ;</u>
<u>100</u>	<u>Indirect with pre-decrement</u>	<u>[Wd--]</u>	<u>[Wd--] = Ws op Wb</u>	<u>- or -</u> <u>Wd <- (Wd - 2) ⁽²⁾ ;</u> <u>EAd = Wd</u> <u>Wd <- (Wd + 1) ⁽¹⁾ ;</u>
<u>101</u>	<u>Indirect with pre-increment</u>	<u>[Wd++]</u>	<u>[Wd++] = Ws op Wb</u>	<u>- or -</u> <u>Wd <- (Wd + 2) ⁽²⁾ ;</u> <u>EAd = Wd</u>
<u>11x</u>	<u>(Unused)</u>			
<u>Note: 1: For byte operations, add or subtract 1.</u>				
<u>2: For word operations, add or subtract 2.</u>				
<u>3: Ws assumed to be in register direct mode (ppp=000).</u>				

TABLE 192 -- Offset Addressing Modes For Wso Source Register (Mode 3)

<u>ggg</u>	<u>Addressing Mode</u>	<u>Source Operand</u>	<u>Effective Address</u>
<u>000</u>	<u>Register Direct</u>	<u>Wns</u>	<u>EA = W register number</u>
<u>001</u>	<u>Indirect</u>	<u>[Wns]</u>	<u>EA = Wns</u> <u>EA = Wns;</u> <u>Wns <- (Wns - 1) ⁽¹⁾</u> <u>- or -</u> <u>Wns <- (Wns - 2) ⁽²⁾</u> <u>EA = Wns;</u> <u>Wns <- (Wns + 1) ⁽¹⁾</u> <u>- or -</u> <u>Wns <- (Wns + 2) ⁽²⁾</u> <u>Wns <- (Wns - 1) ⁽¹⁾;</u> <u>- or -</u> <u>Wns <- (Wns - 2) ⁽²⁾;</u> <u>EA = Wns</u> <u>EA = Wns + Wb ⁽³⁾</u> <u>EA = (Wns + gwwwwww) ⁽⁴⁾</u> <u>- or -</u> <u>EA = (Wns + 2*gwwwwww) ⁽⁵⁾</u>
<u>010</u>	<u>Indirect with post-decrement</u>	<u>[Wns]--</u>	
<u>011</u>	<u>Indirect with post-increment</u>	<u>[Wns]++</u>	
<u>100</u>	<u>Indirect with pre-decrement</u>	<u>[Wns--]</u>	
<u>101</u>	<u>Indirect with register offset</u> <u>Indirect with positive offset by short</u>	<u>[Wns+Wb]</u>	
<u>11g</u>	<u>literal</u> <u>Slit5 ∈ (-16...15)</u>	<u>[Wns+Slit5]</u>	

Note: 1. For byte operations, add or subtract 1.
2. For word operations, add or subtract 2.
3. For byte and word operations, add 2's compliment Wb.
4. For byte operations, add or subtract gwwwwww.
5. For word operations, add or subtract (2 * gwwwwww) or gwwwwww0.

TABLE 193 -- Offset Addressing Modes For Wdo Destination Register (Mode 3)

<u>hhh</u>	<u>Addressing Mode</u>	<u>Source Operand</u>	<u>Effective Address</u>
<u>000</u>	<u>Register Direct</u>	<u>Wnd</u>	<u>EA = W register number</u>
<u>001</u>	<u>Indirect</u>	<u>[Wnd]</u>	<u>EA = Wnd</u> <u>EA = Wnd;</u> <u>Wnd <- (Wnd - 1) ⁽¹⁾</u> <u>- or -</u> <u>Wnd <- (Wnd - 2) ⁽²⁾</u> <u>EA = Wnd;</u> <u>Wnd <- (Wnd + 1) ⁽¹⁾</u> <u>- or -</u> <u>Wnd <- (Wnd + 2) ⁽²⁾</u> <u>Wnd <- (Wnd - 1) ⁽¹⁾;</u> <u>- or -</u> <u>Wnd <- (Wnd - 2) ⁽²⁾;</u> <u>EA = Wnd</u> <u>EA = Wnd + Wb ⁽³⁾</u> <u>EA = (Wnd + hwwwwww) ⁽⁴⁾</u> <u>- or -</u> <u>EA = (Wnd + 2*hwwwwww) ⁽⁵⁾</u>
<u>010</u>	<u>Indirect with post-decrement</u>	<u>[Wnd]--</u>	
<u>011</u>	<u>Indirect with post-increment</u>	<u>[Wnd]++</u>	
<u>100</u>	<u>Indirect with pre-decrement</u>	<u>[Wnd--]</u>	
<u>101</u>	<u>Indirect with register offset</u> <u>Indirect with positive offset by short</u>	<u>[Wnd+Wb]</u>	
<u>11h</u>	<u>literal</u> <u>Slit5 ∈ (-16...15)</u>	<u>[Wnd+Slit5]</u>	

For byte operations, add or subtract 1.
For word operations, add or subtract 2.
For byte and word operations, add 2's compliment Wb.
For byte operations, add or subtract hwwwwww.
For word operations, add or subtract (2 * hwwwwww) or hwwwwww0.

TABLE 194 -- X DATA SPACE PREFETCH OPERATION

<u>iiii</u>	<u>Operation</u>
<u>0000</u>	<u>Wxp=[W4]</u>
<u>0001</u>	<u>Wxp=[W4], W4 = W4 + 2</u>
<u>0010</u>	<u>Wxp=[W4], W4 = W4 + 4</u>
<u>0011</u>	<u>Wxp=[W4], W4 = W4 + 6</u>
<u>0100</u>	<u>No Prefetch for X Data Space</u>
<u>0101</u>	<u>Wxp=[W4], W4 = W4 - 6</u>
<u>0110</u>	<u>Wxp=[W4], W4 = W4 - 4</u>
<u>0111</u>	<u>Wxp=[W4], W4 = W4 - 2</u>
<u>1000</u>	<u>Wxp=[W5]</u>
<u>1001</u>	<u>Wxp=[W5], W5 = W5 + 2</u>
<u>1010</u>	<u>Wxp=[W5], W5 = W5 + 4</u>
<u>1011</u>	<u>Wxp=[W5], W5 = W5 + 6</u>
<u>1100</u>	<u>Wxp=[W5+W8]</u>
<u>1101</u>	<u>Wxp=[W5], W5 = W5 - 6</u>
<u>1110</u>	<u>Wxp=[W5], W5 = W5 - 4</u>
<u>1111</u>	<u>Wxp=[W5], W5 = W5 - 2</u>

TABLE 195 -- X Data Space Prefetch Destination

<u>xx</u>	<u>Wxp</u>
<u>00</u>	<u>W0</u>
<u>01</u>	<u>W1</u>
<u>10</u>	<u>W2</u>
<u>11</u>	<u>W3</u>

TABLE 196 -- MAC or MPY Source Operands

<u>mmm</u>	<u>Multiplicands</u>
<u>000</u>	<u>W0 * W1</u>
<u>001</u>	<u>W0 * W2</u>
<u>010</u>	<u>W0 * W3</u>
<u>011</u>	<u>Invalid (CLRAC instruction)</u>
<u>100</u>	<u>W1 * W2</u>
<u>101</u>	<u>W1 * W3</u>
<u>110</u>	<u>W2 * W3</u>
<u>111</u>	<u>Invalid (MOVS instruction)</u>

TABLE 197 -- Y Data Space Prefetch Operation

<u>iiii</u>	<u>Operation</u>
<u>0000</u>	<u>Wyp=[W6]</u>
<u>0001</u>	<u>Wyp=[W6], W6 = W6 + 2</u>
<u>0010</u>	<u>Wyp=[W6], W6 = W6 + 4</u>
<u>0011</u>	<u>Wyp=[W6], W6 = W6 + 6</u>
<u>0100</u>	<u>No Prefetch for Y Data Space</u>
<u>0101</u>	<u>Wyp=[W6], W6 = W6 - 6</u>
<u>0110</u>	<u>Wyp=[W6], W6 = W6 - 4</u>
<u>0111</u>	<u>Wyp=[W6], W6 = W6 - 2</u>
<u>1000</u>	<u>Wyp=[W7]</u>
<u>1001</u>	<u>Wyp=[W7], W7 = W7 + 2</u>
<u>1010</u>	<u>Wyp=[W7], W7 = W7 + 4</u>
<u>1011</u>	<u>Wyp=[W7], W7 = W7 + 6</u>
<u>1100</u>	<u>Wyp=[W7+W8]</u>
<u>1101</u>	<u>Wyp=[W7], W7 = W7 - 6</u>
<u>1110</u>	<u>Wyp=[W7], W7 = W7 - 4</u>
<u>1111</u>	<u>Wyp=[W7], W7 = W7 - 2</u>

TABLE 198 -- Y Data Space Prefetch Destination

<u>yy</u>	<u>Wyp</u>
<u>00</u>	<u>W0</u>
<u>01</u>	<u>W1</u>
<u>10</u>	<u>W2</u>
<u>11</u>	<u>W3</u>

TABLE 199 -- Mac Accumulator Write Back Selections

<u>aa</u>	<u>Multiplicands</u>
<u>00</u>	<u>W9 = Other Accumulator (direct)</u>
<u>01</u>	<u>[W9]++ = Other Accumulator</u> <u>(indirect, post-increment)</u>
<u>10</u>	<u>No write back</u>
<u>11</u>	<u>Invalid (MPYxxx instruction)</u>

[0525] ~~{0108}~~ While specific embodiments of the invention have been illustrated and described, it will be understood by those having ordinary skill in the art that changes may be made to those embodiments without departing from the spirit and scope of the invention. The invention, therefore, is well adapted to carry out the objects and attain the ends and advantages mentioned, as well as others inherent therein. While the invention has been depicted, described, and is defined by reference to exemplary embodiments of the

invention, such references do not imply a limitation on the invention, and no such limitation is to be inferred. The invention is capable of considerable modification, alternation, and equivalents in form and function, as will occur to those ordinarily skilled in the pertinent arts and having the benefit of this disclosure. The depicted and described embodiments of the invention are exemplary only, and are not exhaustive of the scope of the invention. Consequently, the invention is intended to be limited only by the spirit and scope of the appended claims, giving full cognizance to equivalents in all respects.

Claims

What is claimed is:

1. A processor for executing an instruction set comprising the designated instruction set, the processor comprising: a program memory for storing program instructions including instructions from the designated instruction set; a program counter for determining current instruction for processing; registers for storing operand data specified by the program instructions; and at least one instruction execution unit for executing the current instruction.
2. The processor according to claim 1, wherein the at least one execution unit includes a digital signal processing engine.
3. The processor according to claim 1, wherein the at least one execution unit includes an arithmetic logic unit.
4. The processor according to claim 1, wherein each designated instruction is identified to the processor by the designated encoding.

DIGITAL SIGNAL CONTROLLER MICROCONTROLLER INSTRUCTION SET AND
ARCHITECTURE

Abstract

ABSTRACT OF THE INVENTION

An instruction set is provided that features ~~ninety-four~~ multiple instructions and various address modes to deliver a mixture of flexible ~~micro-controller~~ microcontroller-like instructions and specialized digital signal processor (DSP) ~~processing~~ processing ("DSP") ~~execute~~ instructions that ~~execute~~ from a single instruction stream. A subset of instructions of the instruction set can be executed by a processor. Similarly, another subset of the instructions can be utilized by the digital signal processor. A software application can thus take advantage of digital signal processing capabilities in the same program, obviating the need for separate programs for separate processors.